Java
for
Aliens

Claudio De Sio Cesari

Java for Aliens

Learn Java from scratch and become a Pro

Exercises and Solutions

Java for Aliens - Exercises and Solutions

Copyright © 2019 by **Claudio De Sio Cesari**

Editor: Emanuele Giuliani (**emanuele@giuliani.mi.it**)

First Edition (November, 2019)

Font licenses
Libre Baskerville (**https://fonts.google.com/specimen/Libre+Baskerville**, Impallari Type): OFL
Libre Franklin (**https://fonts.google.com/specimen/Libre+Franklin**, Impallari Type): OFL
Cousine (**https://fonts.google.com/specimen/Cousine**, Steve Matteson): AL
Inconsolata (**https://fonts.google.com/specimen/Inconsolata**, Raph Levien): OFL
Roboto (**https://fonts.google.com/specimen/Roboto**, Christian Robertson): AL
Digits (**https://www.1001fonts.com/digits-font.html**, Dieter Steffmann): FFC
Journal Dingbats 3 (**https://www.1001fonts.com/journal-dingbats-3-font.html**, Dieter Steffmann): FFC
Musicals (**https://www.1001fonts.com/musicals-font.html**, Brain Eaters): FFC

Image licenses
Curiosity icon (**https://www.flaticon.com/free-icon/toyger-cat_107975**, www.freepik.com): FBL
Alien icon (**http://www.iconarchive.com/show/free-space-icons-by-goodstuff-no-nonsense/alien-4-icon.html, goodstuffnononsense.com**): CC
Trick icon (**https://www.flaticon.com/free-icon/magic-wand_1275106**, www.flaticon.com/authors/pause08): FBL

License specifications
Open Free License (OFL): https://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL_web
Apache License, Version 2.0 (AL): http://www.apache.org/licenses/LICENSE-2.0
1001Fonts Free For Commercial Use License (FFC): https://www.1001fonts.com/licenses/ffc.html
Flaticon Basic License (FBL): https://file000.flaticon.com/downloads/license/license.pdf
CC Attribution 4.0 (CC): https://creativecommons.org/licenses/by/4.0/legalcode

Table of Contents

Table of Contents

Java for Aliens Exercises

Introduction to Exercises

This document represents, together with the Appendices document, the natural completion of the book "Java for Aliens".

All the exercises have been moved to this document so as not to take space away from the theory and not to increase the cost of the book.

For each chapter of the book (and for each appendix, where applicable) specific exercises have been designed to validate (and expand) what has been learned during the study.

The exercises are fundamental, essential. The theory is usually clear, but applying the concepts learned is far from easy. Developing not only includes the implementation of the code, but many other components are part of it, and they influence the final result.

The exercises that you will find in this document therefore strive to insist on topics that other books treat superficially, or do not deal with at all, such as analysis, design and object-oriented architecture. There are also exercises for all subjects, even for those that may seem trivial to the more experienced reader, but which may be fundamental for the neophyte. In this book, you will find over 600 exercises!

> **Warning! This document will be updated in the future. Currently there are some exercises not yet published since they need some further revisions. Updates will allow us to carry out further checks on the quality and correctness of the content, and add more exercises. The author will notify any news or information via his social media channels.**

Based on reader feedback, this file has been organized and improved clarity and the style, and created exercises that can satisfy all types of readers. There are exercises in which it is required to add parts of code to fix bugs, code algorithms, design simple applications, or create more complex applications step by step. Dozens of single or multiple-choice exercises have also been introduced. They are similar to the test exercises of the Oracle Java programming certification exams. These exercises help in preparing for certifications, and have been set up in such a way that the programmer gains confidence. They often consist, essentially, of reading the code and understanding its meaning and details, and also represent a remarkable test bench for the most expert readers. I'm sure that the reader will appreciate the effort made to achieve such a large number of heterogeneous and original exercises.

Unlike many other texts, we have provided solutions for all the exercises (with few exceptions). Obviously, when it comes to coding a solution, there are hundreds of valid alternatives, so we must not consider the proposed solutions as, objectively, the better ones. You can download all the list of solutions and exercises (together with all the code examples included in the text) at the same address where you have downloaded the file you are reading: **http://www.javaforaliens.com.**

Let's remember once again that, especially for those who are beginners, it is important to start writing all the code by hand, without using copy-paste or special help from the development tool chosen. It is also very important to use comments for all of our code. This will allow us to learn the definitions better, and to have more security when writing code.

So, it is fundamental to write the source code on a text editor such as Windows Notepad (as described in the first chapter) and compile using the command line (see Appendix A). We do not advise performing the exercises of these very first chapters (let's say the first 4) using a complex IDE like Eclipse or Netbeans… we could end up studying the IDE rather than Java.

It is, instead, advisable (if you do not want to have too much to do with Notepad and the command line) to use EJE (**https://sourceforge.net/projects/eje**) which offers simple utilities designed for those who start to program. For example, it allows us to compile and execute our files by pressing two distinct buttons.

> **EJE is easy to install (just unzip the file in any folder) and use. However, Appendix M is dedicated to its description.**

Finally we suggest, after having done an exercise, to consult the relative solution before moving on to the next one (often an exercise is a prerequisite for the next one).

For any type of communication, you can write directly to the author at: **claudio@claudiodesio.com**. You can also contact him via the most important social networks and through his personal

website (also to keep up to date with news, the Telegram group will notify you only for news concerning this book):

- ■ Telegram: **http://t.me/java4aliens**
- ■ Facebook: **http://www.facebook.com/claudiodesiocesari**
- ■ Twitter: **http://twitter.com/cdesio**
- ■ LinkedIn: **http://www.linkedin.com/in/claudiodesio**
- ■ Internet: **http://www.claudiodesio.com**
- ■ YouTube: **http://www.youtube.com/claudiodesiocesari**
- ■ Instagram: **http://www.instagram.com/cdesio**

Happy working!

Claudio De Sio Cesari

Chapter 1
Exercises

Introduction to Java

The following exercises are designed for those starting from scratch. These exercises have the goal of giving a minimum of confidence with the Java programming environment to the reader.

Let's remember once again that, especially for those who are beginners, it is important to start writing all the code by hand, without copy-paste or special help from the development tool chosen. It is also very important to use comments for every line of code written. This will allow us to learn better the definitions, and to be more aware when writing code.

So, it will be fundamental to write the source code on a text editor such as Windows Notepad (as described in the first chapter) and compile using the command line (see Appendix C). We do not recommend performing the exercises of these very first chapters (let's say the first 4) using a complex IDE like Eclipse or Netbeans... we could end up studying the IDE rather than Java.

It is instead advisable (if you do not want to have too much to do with Notepad and the command line) to use EJE (**https://sourceforge.net/projects/eje**), which offers simple features designed for helping the beginners to start to code. For example, it allows us to compile and execute our files by pressing two distinct buttons.

> **EJE must be downloaded at** https://sourceforge.net/projects/eje. **It is easy to install (just unzip the file in any folder) and use. To run it, double-click on the** `eje.bat` **file. However, you can find more information in Appendix M.**

From Chapter 5 onwards, it will be easier to switch to an IDE, since other development environments will be explained.

Finally we suggest, after having done an exercise, to consult the relative solution before moving on to the next one (often an exercise is a prerequisite for the next one). This advice applies to all exercises in all chapters and appendices.

At the same address (**http://www.javaforaliens.com/download.html**) where you downloaded the appendices you can also download all the sample code contained in the book and in the appendices (always downloadable at the same address), and all the code of the exercises (with solutions) related to all the chapters and all the appendices.

Exercise 1.a)

Write, save, compile and run the `HelloWorld` program. We recommend to the reader to do this exercise twice: the first time using Notepad and the DOS prompt, and the second using EJE.

> **EJE allows us to insert pre-formatted parts of code via the `Insert` menu (or using shortcuts).**

Exercise 1.b) Basic Concepts of Computer Science, True or False:

1. A computer is composed of hardware and software.

2. The operating system is part of a computer's hardware. In fact, a computer cannot function without an operating system.

3. Windows Notepad is software.

4. The power supply cable of a computer is hardware.

5. Machine language is the language that a computer processor can interpret.

6. The machine language is unique and standard.

7. The machine language has a vocabulary that contains only two symbols: `0` and `1`.

8. Both the compiler and the interpreter have the task of translating the written instructions with a certain programming language into machine language instructions.

9. In general, a program written in an interpreted language has a faster execution time than a program written in a compiled language.

10. An executable program is composed of its source files.

2

Exercise 1.c) Features of Java, True or False:

1. Java is the name of a technology and at the same time the name of a programming language.

2. Java is an interpreted language but not a compiled language.

3. Java is a language fast but not robust.

4. Java is a difficult language to learn because in any case it forces you to learn Object Orientation too.

5. The Java Virtual Machine is a software that supervises execution of the software written in Java.

6. The JVM handles memory automatically through garbage collection.

7. Platform independence is an unimportant feature.

8. A Java system is a closed system.

9. Garbage collection guarantees platform independence.

10. Java is a free language that collects the best features of other languages, and excludes those deemed worse and more dangerous.

Exercise 1.d) Java Code, True or False:

1. The following declaration of the `main()` method is valid:

```
public static main(String arguments[]) {...}
```

2. The following declaration of the `main()` method is valid:

```
public static void Main(String args[]){...}
```

3. The following declaration of the `main()` method is valid:

```
public static void main(String arguments[]) {...}
```

4. The following declaration of the `main()` method is valid:

```
public static void main(String Arguments[]) {...}
```

5. The following class declaration is correct:

```
public class {...}
```

3

6. The following class declaration is correct:

```
public Class Car {...}
```

7. The following class declaration is correct:

```
public class Car {...}
```

8. You can declare a method outside the block of code that defines a class.

9. The block of code that defines a method is delimited by two round brackets.

10. The block of code that defines a method is delimited by two square brackets.

Exercise 1.e) Development Environment and Process, True or False:

1. The JVM is software that simulates hardware.

2. The bytecode is contained in a file with the suffix **.class**.

3. Java development consists of writing the program, saving it, running it and finally compiling it.

4. Java development consists of writing the program, saving it, compiling it and finally running it.

5. The name of the file that contains a Java class must match the name of the class, regardless of whether the letters are uppercase or lowercase.

6. Once you have compiled a program written in Java, you can run it on any operating system that has a JVM.

7. To run any Java application, all you need is a browser.

8. The JDK compiler is invoked via the **javac** command and the JVM is invoked via the **java** command.

9. To run a file called **Foo.class**, we need to run the following command from the prompt: `java Foo.java`.

10. To run a file called **Foo.class**, we need to run the following command from the prompt: `java Foo.class`.

Exercise 1.f)

Delete the `static` modifier of the `main()` method from the `HelloWorld` class. Using the DOS

4

prompt, compile and run the program, and interpret the execution error message.

> **Knowing how to interpret error messages is absolutely essential.**

Exercise 1.g)

Delete the first open brace encountered by the `HelloWorld` class. Using the DOS prompt, compile the program and interpret the compilation error message.

Exercise 1.h)

Delete the last closed parenthesis (last symbol of the program) from the `HelloWorld` class. Using the command line, compile the program and interpret the error message.

Exercise 1.i)

Delete the symbol " ; " from the `HelloWorld` class. Using the DOS prompt, compile the program and interpret the error message.

Exercise 1.j)

Double the closing brace of the `HelloWorld` class. Using the DOS prompt, compile the program and interpret the error message

Exercise 1.k)

Add a block of braces inside the `main()` method:

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

Compile, execute and draw your conclusions.

Exercise 1.l)

Double the symbol " ; " in the `HelloWorld` program, then compile and execute it. What happens?

5

Exercise 1.m)

Write the `HelloWorld` program by writing each word and every symbol on the next line. What is the problem?

Exercise 1.n)

Change the `HelloWorld` program in order to print another string instead of the "Hello World!" string.

Exercise 1.o)

Make the `HelloWorld` program print a number instead of the "Hello World!" string.

Exercise 1.p)

Try printing the sum of two numbers to the `HelloWorld` program instead of the "Hello World!" string, after reading the solution from the previous Exercise 1.o.

Exercise 1.q)

Compile and run the following program:

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

    }
}
```

what is the output?

Exercise 1.r)

Compile and run the following program:

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

what is the output?

Exercise 1.s)

Compile and run the following program:

6

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println(args);
    }
}
```

what is the output?

Exercise 1.t)

Write a program named `ShoppingList` that prints a shopping list, where each item to buy resides on its own line.

Exercise 1.u)

Write a program named `CompactShoppingList` that prints a shopping list, where each item to buy is separated from another with a comma.

Exercise 1.v)

Create a new file named `SayJava` that prints out the string "JAVA" as in the following example.

```
--------
| JAVA |
--------
```

Exercise 1.w)

Write a program defined by the `Arrows` class that prints the following output:

```
<----<<<

>>>---->
```

Exercise 1.x)

Write a program defined by the `PrintContacts` class that prints the contact list of a phone book. Each contact must be printed on three lines: in the first there will be the name of the contact, in the second the address, and in the third the phone number. Each contact must be separated from the next one by an empty line.

Exercise 1.y)

Create a class called `PrintEmptyRowClass` that prints the following output:

```
public class EmptyRow{
    public static void main(String args[]) {
        System.out.println();
    }
}
```

Exercise 1.z)

Create a new file named `SayMyName` that prints out your name as in the following example:

```
*****   *       ******   *    *   **      *    *****
*       *       *    *    *    *   *  *    *    *   *
*       *       ******    *    *   *   *   *    *   *
*       *       *    *    *    *   *  *    *    *   *
*****   *****   *    *    *****    ***     *    *****
```

> **Note that to print each character 5 columns and 5 rows were used (except for the `I` character). Each character is separated from another one by 3 columns.**

Chapter 1
Exercise Solutions

Introduction to Java

To solve our exercises, there are often hundreds of different and valid solutions. Each of them has its pros and cons. Therefore, we must not take the solution proposed for an exercise as the only existing one. This concept also applies to all other chapters. This does not apply to solutions that do not consist of code (for example "True or False" exercises).

Solution 1.a)

The code could be the following:

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

For the instructions to be performed from the command line to compile and execute the code, see sections 1.3.2 and 1.4.3.

Solution 1.b) Basic Concepts of Computer Science, True or False:

1. **True**.

2. **False**, the operating system is software.

3. **True**.

4. **True**.

5. **True**.

6. **False**, each processor defines its own machine language.

7. **True**.

8. **True**.

9. **False**, because an interpreted language must alternate the translation phase with the execution phase, so it is usually slower.

10. **False**, an executable program is composed of its binary files.

Solution 1.c) Java Features, True or False:

1. **True**.

2. **False**.

3. **False**.

4. **True**.

5. **True**.

6. **True**.

7. **False**.

8. **False**.

9. **False**.

10. **True**.

Solution 1.d) Java Code, True or False:

1. **False**, the return type (`void`) is missing.

2. **False**, the identifier (`main`) should start with a lowercase letter.

3. **True**.

4. **True**.

5. **False**, the identifier is missing.

6. **False**, the keyword (`class`) must be written with a lowercase letter.

7. **True**.

8. **False**.

9. **False**, round brackets must be braces.

10. **False**, square brackets must be braces.

Solution 1.e) Development Environment and Process, True or False:

1. **True**.

2. **True**.

3. **False**, you must first compile it and then send it running.

4. **True**.

5. **False**, we must also take capital and small letters into account.

6. **True**.

7. **False**, one browser is sufficient only to run applets (which today are no longer supported).

8. **True**.

9. **False**, the right command is **java Foo**.

10. **False**, the right command is **java Foo**.

Solution 1.f)

The code should be the following:

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

The file quietly compiles, but at runtime we'll be warned that we have defined a `main()` method which is not a valid method for starting the application, precisely because it has not been de-

clared with the `static` modifier:

```
Error: Main method is not static in class HelloWorld, please define the main method as:
   public static void main(String[] args)
```

Solution 1.g)

The code should be similar to the following:

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

The compiler error message is as follows:

```
error: '{' expected
public class HelloWorld
                        ^
1 error
```

Solution 1.h)

The code should be similar to the following:

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

The compiler error message is as follows:

```
error: reached end of file while parsing
    }
     ^
1 error
```

which tells us that the end of the file has been reached... and something is missing.

Solution 1.i)

The code should be similar to the following:

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

```
            System.out.println("Hello World!")
        }
    }
```

The compiler error message is as follows:

```
error: ';' expected
        System.out.println("Hello World!")
                                          ^
1 error
```

Where the compiler warns us that a semicolon is missing.

Solution 1.j)

The error message is as follows:

```
HelloWorld.java:5: error: class, interface, or enum expected
}
^
1 error
```

The message is the same as we have already seen in section 1.5.1, where we had defined a class by mistakenly using the capital letter for the class keyword. In that case, as we know, a Java source file must necessarily define a class within it. The compiler since it had not found a valid definition, claimed the definition of a class (or an interface or an enumeration, but these last two concepts have not yet been defined). In this case instead, the compiler expects that in place of the superfluous brace, another class (or an interface or an enumeration) will be defined. In fact, as we will see later, it is possible to define other classes within a single source file

Solution 1.k)

The file is compiled and executed as if the pair of braces did not exist. In fact, it is possible to use pairs of braces within our methods, perhaps surrounding other instructions. For example we could also write:

```
public class HelloWorld {
    public static void main(String args[]) {
        {
            System.out.println(args);
        }
    }
}
```

Also here, the braces are undoubtedly superfluous, but there are rare cases in which the parenthesis can be used to isolate pieces of code, from the rest. For now, we just need to know that braces, if used in pairs, can be used within our source files.

13

Solution 1.l)

The code should be similar to the following:

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

However, the file is compiled and executed without errors. In fact, the superfluous ";" symbol is considered by the compiler as a (legal) termination of an empty statement. We could also write it on the next line (since as we read in Chapter 1 nothing changes) to "see" the empty statement:

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

Solution 1.m)

The code should be similar to the following:

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

```
    )
    ;
    }
}
```

However, the file is compiled and executed without errors. In fact, as we will see in the next chapters, Java is a free form language. The problem is that it becomes very complicated to read for us.

Solution 1.n)

The code could be similar to the following:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Una frase a piacere!");
    }
}
```

Solution 1.o)

The code could be similar to the following:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("8");
    }
}
```

but we printed the number as a string (the strings will be the subject of the third chapter), in fact we enclosed it in two quotation marks. We could also write directly:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println(8);
    }
}
```

This time we are printing a different data type (there are no double quotes). In the next exercise we will begin to understand the situation better.

Solution 1.p)

The code could be similar to the following:

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

```
            System.out.println("25+7");
    }
 }
```

The output therefore will not print a sum, but simply:

```
25+7
```

We could also write this without double quotes:

```
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println(25+7);
    }
 }
```

In this case the output will be the desired one:

```
32
```

In fact, the **numeric data types** (which are written without double quotes as we will see in Chapter 3), allow us to perform arithmetic operations.

Solution 1.q)

The program does not print anything because the print instruction is missing (System.out.println()).

Solution 1.r)

The program does not print anything because nothing is printed in the print instruction System.out.println(). Note, however, that the cursor has dropped to the next line, due to the fact that the instruction System.out.println() always wraps after printing (but also after not printing). In fact, println stands for "print line".

Solution 1.s)

In this case the output will be similar to the following:

```
[Ljava.lang.String;@5679c6c6
```

An object (called args) has been "printed" and we will understand in the next few chapters why it has such a mysterious string representation.

Solution 1.t)

The code should be similar to the following:

```java
public class ShoppingList {
    public static void main(String args[]) {
        System.out.println("bread");
        System.out.println("coffee");
        System.out.println("tea");
        System.out.println("fruit");
    }
}
```

E l'output è il seguente:

```
bread
coffee
tea
fruit
```

Solution 1.u)

The code should be similar to the following:

```java
public class CompactShoppingList {
    public static void main(String args[]) {
        System.out.println("bread, coffee, tea, fruit");
    }
}
```

And the output is the following:

```
bread, coffee, tea, fruit
```

Solution 1.v)

The code could be similar to the following:

```java
public class SayJava {
    public static void main(String args[]) {
        System.out.println("----------");
        System.out.println("| JAVA |");
        System.out.println("----------");
    }
}
```

Solution 1.w)

The required code should be the following:

```java
public class Arrows {
    public static void main(String args[]) {
        System.out.println("<----<<<");
        System.out.println("");
        System.out.println(">>>---->");
    }
}
```

To print an empty line, the natural solution was to use the `System.out.println("")` statement, which passes an empty string `""` as a parameter to the `println()` method. Actually, it is also possible to use the `System.out.println()` statement instead, without passing any parameter to the method.

Solution 1.x)

The required code should be similar to the following:

```java
public class PrintContacts {
    public static void main(String args[]) {
        System.out.println("Contacts List");
        System.out.println();
        System.out.println("Claudio De Sio Cesari");
        System.out.println("13, Java Street");
        System.out.println("131313131313");
        System.out.println();
        System.out.println("Stevie Wonder");
        System.out.println("10, Music Avenue");
        System.out.println("1010101010");
        System.out.println();
        System.out.println("Gennaro Capuozzo");
        System.out.println("1, Four Days of Naples Square");
        System.out.println("1111111111");
    }
}
```

Note that to print blank lines, we used the `System.out.println()` method without passing any parameters. The solution would have been valid even if we had used an empty string like this: `System.out.println("")`.

Solution 1.y)

The required code should be the following:

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

Note that we did not print anything to the System.out.println() method, because otherwise we would have had problems with the double quotes. In fact, if we had written:

```
System.out.println("        System.out.println("");");
```

we would get the following error:

```
PrintEmptyRowClass.java:5: error: ')' expected
        System.out.println("        System.out.println("");");
                                                         ^
1 error
```

This is because the compiler cannot understand that the second double quotes it encounters must be considered to be printed, and not as double quotes that close a string! The second double quotes are therefore considered to be the closing double quotes of the string highlighted in bold:

```
System.out.println("        System.out.println("");");
```

So the third double quotes is not accepted, because the compiler expects the parenthesis of the println() method to be closed. We will see in the next chapters how to solve this problem.

Solution 1.z)

The code could be similar to the following:

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

Chapter 2
Exercises

Key Components of a Java Program

Remember that many exercises are preparatory to the following ones, so we recommend doing all the exercises or at least consulting the solutions before going ahead.

Exercise 2.a)

The following class is provided (copy, save and compile):

```java
public class IntegerNumber {
    public int integerNumber;
    public IntegerNumber() {
    }
    public void printNumber() {
        System.out.println(integerNumber);
    }
}
```

This class defines the concept of an integer as an object. It declares an integer variable and a method that will print the variable itself.

Write, compile and execute a class that:

- ▪▪ will instantiate at least two objects from the `IntegerNumber` class (containing a `main()` method);

- ▪▪ will change the value of the corresponding variables and test that these values are correctly assigned, calling the method `printNumber()` on the two objects;

- ▪▪ will add a constructor to the `IntegerNumber` class to initialize the instance variable.

Two more questions:

- ▪▪ what type of variable is the `integerNumber` variable defined in the `IntegerNumber` (local variable, parameter or instance variable)?

- ▪▪ If we instantiate an object of the `IntegerNumber` class, without assigning a new value to the integer variable, what will be the value of the latter?

Exercise 2.b) Key Components Concepts, True or False:

1. An instance variable must necessarily be initialized by the programmer.
2. A local variable shares the life cycle with the object in which it is defined.
3. A parameter has a life cycle coinciding with the method in which it is declared: it is created when the method is invoked, it is not more usable when the method ends.
4. An instance variable belongs to the class in which it is declared.
5. A method is synonymous with action, operation.
6. Both variables and methods are usually usable through the dot operator, applied to an instance of the class where they were declared.
7. A constructor is a method that never returns anything, in fact it has `void` return type.
8. A constructor is called the "default constructor" , if it has no parameters.
9. A constructor is a method and therefore can be used through the dot operator, applied to an instance of the class where it was declared.
10. A package is physically a folder containing classes, which explicitly declared to be part of the package itself in the respective source files.

Exercise 2.c) Key Components Syntax, True or False:

1. In a method declaration (not constructor), the name is always followed by brackets surrounding the optional parameters, and is always preceded by a return type.

2. The following method is correctly defined:

```
public void method() {
    return 5;
}
```

3. The following method is correctly defined:

```
public int method() {
    System.out.println("Ciao");
}
```

4. The following variable is correctly defined:

```
public int a = 0;
```

5. The following variable x is correctly used (refer to the Point class defined in this chapter):

```
Point p1 = new Point ();
Point.x = 10;
```

6. The following variable x is correctly used (refer to the Point class defined in this chapter):

```
Point p1 = new Point();
Point.p1.x = 10;
```

7. The following variable x is correctly used (refer to the Point class defined in this chapter):

```
Point p1 = new Point();
x = 10;
```

8. The following constructor is correctly used (refer to the Point class defined in this chapter):

```
Point p1 = new Point();
p1.Point();
```

9. The following constructor is correctly defined:

```
public class Computer {
    public void Computer(){
    }
}
```

10. The following constructor is correctly defined:

```
public class Computer {
    public computer(int a) {

    }
}
```

23

Exercise 2.d)

Create a Square class, which declares a side instance variable of type int. Then create a public method called perimeter() that returns the perimeter of the square, and a public area() method that returns the area of the square.

> **Remember that the perimeter is the sum of the sides of the square, while the area is calculated by multiplying the side by itself. Finally, the symbol to perform a multiplication in Java is the asterisk *.**

Exercise 2.e)

Create a SquareTest class that contains a main() method that instantiates an object of type Square, with side of value 5. Then print the perimeter and the area of the object just created.

Exercise 2.f)

After doing the previous exercise, you should have set the variable side with a statement like the following:

```
objectName.side = 5;
```

To avoid to write this statement, create a constructor in the Square class of the Exercise 2.d, which takes the value of the variable side as input. Once done, compile the Square class. The SquareTest class, on the other hand, will no longer compile due to the instruction specified above and the non-use of the new constructor. Modify the code of the SquareTest class so that it compiles and runs correctly.

Exercise 2.g)

In the Square class created in the Exercise 2.d, replace the value 4 used to calculate the perimeter, with an instance constant named SIDES_NUMBER.

> **Note that for the constant, a name consisting of only uppercase letters separated with an underscore symbol was used. This is a convention that is used for all constants as explained in section 3.1.**

This should not affect the SquareTest class.

24

Exercise 2.h)

Create a `Rectangle` class equivalent to the `Square` class created in the Exercise 2.d and refined in subsequent exercises. Before coding the class, decide which variables and methods this class must have.

> **Never code directly. It's a classic mistake that can lead to getting lost when you're just start learning programming. You first need to define the specifications in your mind, or even better on a sheet of paper. The advice is to have clear the various definitions (instance variables, local variables, methods, parameters, constructors, etc.).**

Exercise 2.i)

Create a `RectangleTest` class that contains a `main()` method and that tests the `Rectangle` class, equivalently as we did in Exercise 2.e. This time, at least two different rectangles must be instantiated.

Exercise 2.j)

Add to both `Square` and `Rectangle` classes created in the previous exercises, a method called `printDetails()`, which prints the details of the geometric figure, including perimeter and area. Also create a new version of the `SquareTest` and `RectangleTest` classes that directly invoke the `printDetails()` methods on the instantiated objects

Exercise 2.k)

Starting from the solution of the previous exercise, create an additional method in the `Square` and `Rectangle` classes, called `getDetails()`, which returns the same string that was printed in the `printDetails()` method. After creating it, make sure that the `printDetails()` method takes advantage of the `getDetails()` method so as not to duplicate the code. Create a class called `TestQuadrilaters` that prints the details of a square and a rectangle.

Exercise 2.l)

Abstract the concept of `Nation` with a class, creating at least one constructor and instance variables, but no methods.

Exercise 2.m)

After creating the `Nation` class of Exercise 2.l, can you create one or more methods? If you can, define them within the class. If you can't, can you explain why?

Exercise 2.n)

Given the following class:

```
public class Exercise2N {
    public String string;
    public int integer;
    final public String INTEGER = "initialization";
}
```

Which of the following statements is true (choose only one statement)?

1. The code can be compiled correctly.

2. The code cannot be compiled correctly because it is not possible to declare a variable with the name `string`.

3. The code cannot be compiled correctly because it is not possible to declare a variable with the name `integer`.

4. The code cannot be compiled correctly because it is not possible to declare a variable of type `String` by calling it `INTEGER`.

5. The code cannot be compiled correctly because the variable with the name `integer` declares the modifiers in reverse order (it should first be declared `public` and then `final`).

Exercise 2.o)

Given the following class:

```
public class Exercise2O {
    public String toString() {
        return "Exercise2O";
    }

    public void main() {

    }

    public void static method() {

    }
```

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

    }
}
```

There is only one error in this class that will prevent it from being compiled, which one?

> **If you are unable to answer the question, write the class by hand, compile and interpret the error. Then fix it and compile the file.**

Exercise 2.p)

Given the following class:

```
public class Exercise2P {
    public String string;

    public static void method(String arguments[]) {
        public int integer=0;
    }
}
```

There is only one error in this class that will prevent it from being compiled, which one?

> **If you are unable to answer the question, write the class by hand, compile and interpret the error. Then fix it and compile the file.**

Exercise 2.q)

Given the following class:

```
public class Exercise2Q {

    public static void main(String arguments) {
        System.out.println("Quelo")
    }
  }
}
```

There are three errors in this class that will prevent it from being compiled, which ones?

> **If you are unable to answer the question, write and complete the class by hand, and interpret the errors. Then fix the errors and compile the file.**

Exercise 2.r)

Given the following class:

```
public class Exercise2R {
    public int var1;
    public int var2;

    System.out.println("Exercise 2.r");

    public Exercise2R() {

    }

    public Exercise2R(int a , int b) {
        var1 = b;
        var2 = a;
    }

    public static void main(String args[]) {
        Exercise2R exercise2R = new Exercise2R (4,7);
        System.out.println(exercise2R.var1);
        System.out.println(exercise2R.var2);
    }
}
```

Once executed, what will this program print?

1. This program cannot be run.

2. This program does not compile.

3. It will print 74.

4. It will print 47.

Exercise 2.s)

Given the following classes:

```
public class Course {
    public String name;

    public Course() {

    }

    public Course(String n) {
        name = n;
```

```
        }
    }

public class Exercise2S {

    public static void main(String args[]) {
        Course course1 = new Course();
        course1.name = "Java";
        Course course2 = new Course("Java");
        System.out.println(course1.name);
        System.out.println(course2.name);
    }
}
```

Which sequence of instructions among the following is needed to execute the program?

1. `javac Course.java, javac Exercise2S.java, java Course`

2. `javac Course.java, javac Exercise2S.java, java Course.class`

3. `javac Course.java, javac Exercise2S.java, java Exercise2S`

4. `javac Course.java, javac Exercise2S.java, java Exercise2S Course`

Exercise 2.t)

Given the following classes:

```
public class Course {
    public String name;

    public Course() {

    }

    public Course(String n) {
        name = n;
    }
}

public class Exercise2T {

    public static void main(String args[]) {
        Course course1 = new Course();
        course1.name = "Java";
        Course course2 = new Course("Java");
        System.out.println(course1.name);
        System.out.println(course2.name);
    }
}
```

Once executed, what this program will print?

1. This program cannot be run.

2. This program does not compile.

3. Will print:

```
Java
Java
```

4. Will print:

```
Java
```

Exercise 2.u)

Given the following class:

```java
public class Exercise2U {
    int c = 3;
    public static void main(String args[]) {
        int a = 1;
        int b = 2, c, d = 4;
        System.out.println(a+b+c+d);
    }
}
```

Once executed, what this program will print?

1. This program does not compile.

2. Will print:

```
10
```

3. Will print:

```
7
```

4. Will print:

```
0
```

Exercise 2.v)

Create a class called Exercise2V that allows you to get the sum of 2, 3, 5 and 10 integers.

Exercise 2.w)

Consider the solution in Exercise 1.x, where we created a class that simulated the printing of the details of some contacts in a phone book:

```java
public class PrintContacts {
    public static void main(String args[]) {
        System.out.println("Contacts List");
        System.out.println();
        System.out.println("Claudio De Sio Cesari");
        System.out.println("13, Java Street");
        System.out.println("131313131313");
        System.out.println();
        System.out.println("Stevie Wonder");
        System.out.println("10, Music Avenue");
        System.out.println("1010101010");
        System.out.println();
        System.out.println("Gennaro Capuozzo");
        System.out.println("1, Four Days of Naples Square");
        System.out.println("1111111111");
    }
}
```

Abstract, save, and compile and a `Contact` class that contains the necessary variables and one or more constructors.

Exercise 2.x)

Consider the solution of the Exercise 2.w, create a new version of the `PrintContacts` class of the Exercise 1.x (whose code is also reported in the Exercise 2.w), this time taking advantage of the `Contact` class created in the Exercise 2.w. The output of the program must be the same as the program `PrintContacts` of the Exercise 1.x.

> **Tip: use a method similar to `printDetails()` that we have defined in the solution of Exercise 2.k.**

Exercise 2.y)

Considering the solution in Exercise 2.x, create a PhoneBook class, which contains the contacts created in Exercise 2.x. It must define a constructor without parameters that instantiate its own instance variables. Create a new version of the `PrintContacts` class, which always has the same output. This version must not instantiate the `Contact` objects, but retrieve them from the PhoneBook class.

Exercise 2.z)

Create a `City` class that abstracts the concept of city. Then declare a `Nation` class declaring a `capital` instance variable of type `City`. Finally, create an `Exercise2Z` class that creates a nation with a capital, and prints a sentence that verifies the actual association between the nation and the capital.

We recommend that you create constructors for these classes.

Chapter 2
Exercise Solutions

Key Components
of a Java Program

A class that complies with the requirements is listed below:

```java
public class RequestedClass {
    public static void main (String args []) {
        IntegerNumber one = new IntegerNumber();
        IntegerNumber two = new IntegerNumber();
        one.integerNumber = 1;
        two.integerNumber = 2;
        one.printNumber();
        two.printNumber();
    }
}
```

Furthermore, a constructor for the `IntegerNumber` class could set the only instance variable `integerNumber`:

```java
public class IntegerNumber {
    public int integerNumber;

    public IntegerNumber() {
    }
```

```
    public IntegerNumber(int n) {
        integerNumber = n;
    }

    public void printNumber() {
        System.out.println(integerNumber);
    }
}
```

In this case, however, to instantiate objects from the `IntegerNumber` class, it will no longer be possible to use the default constructor (which will no longer be inserted by the compiler). So, the following statement would produce a compilation error.

```
IntegerNumber one = new IntegerNumber();
```

Instead, create objects by passing the value to the constructor, for example:

```
IntegerNumber one = new IntegerNumber(1);
```

Answers to the two questions:

1. This is an instance variable, because declared within a class, outside of methods.

2. The value will be zero, which is the null value for an integer variable. In fact, when an object is instantiated, instance variables are initialized to null values if not explicitly initialized to other values.

Solution 2.b) Key Components Concepts. True or False:

1. **False**, a local variable must necessarily be initialized by the programmer.

2. **False**, an instance variable shares the life cycle with the object to which it belongs.

3. **True**.

4. **False**, an instance variable belongs to an object instantiated by the class in which it is declared.

5. **True**.

6. **True**.

7. **False**, a constructor is a method that never returns anything, in fact it has no return type.

8. **False**, a constructor is called the "default constructor" if it is inserted by the compiler. It also has no parameters, but not all constructors without parameters are default constructors.

34

9. **False**, a constructor is a special method that has the characteristic of being invoked once and only once an object is instantiated.

10. **True**.

Solution 2.c) Key Components Syntax. True or False:

1. **True**.

2. **False**, it attempts to return an integer value but declares void as return type.

3. **False**, the method should return an integer value.

4. **True**.

5. **False**, the dot operator must be applied to the object and not to the class:

```
Point p1 = new Point();
p1.x = 10;
```

6. **False**, the dot operator must be applied to the object and not to the class, furthermore the class does not "contain" the object.

7. **False**, the dot operator must be applied to the object. The compiler would not find the declaration of the x variable.

8. **False**, a constructor is a special method that has the characteristic of being invoked once and only once an object is instantiated, using the new operator.

9. **False**, the constructor does not declare a return type and must have a name coinciding with the class.

10. **False**, the constructor must have a name coinciding with the class.

Solution 2.d)

The code should be similar to the following:

```
public class Square {
    public int side;

    public int perimeter() {
        int perimeter = side * 4;
        return perimeter;
    }
}
```

```
    public int area() {
        int area = side * side;
        return area;
    }
}
```

Solution 2.e)

The code should be similar to the following:

```
public class SquareTest {
    public static void main(String args[]) {
        Square square = new Square();
        square.side = 5;
        int perimeter = square.perimeter();
        System.out.println(perimeter);
        int area = square.area();
        System.out.println(area);
    }
}
```

Note that we have created the `perimeter` and `area` local variables with the same name as the method, and this is not a problem. In fact, the name of a method always differs from the name of a variable because it is declared with round brackets. We could also have called the variables differently, but it is a good practice that the names are self-explanatory. However, we could also completely avoid the use of these variables if we had written the class like this:

```
public class SquareTest {
    public static void main(String args[]) {
        Square square = new Square();
        square.side = 5;
        System.out.println(square.perimeter());
        System.out.println(square.area());
    }
}
```

The code is more compact, but at least at the beginning, it is better to use the variables to better memorize the definitions.

Solution 2.f)

The `Square` class code should look like the following:

```
public class Square {
    public int side;
```

```
    public Square(int l) {
        side = l;
    }
    public int perimeter() {
        int perimeter = side * 4;
        return perimeter;
    }

    public int area() {
        int area = side * side;
        return area;
    }
}
```

The `SquareTest` class code should look like the following:

```
public class SquareTest {
    public static void main(String args[]) {
        Square square = new Square(5);
        int perimeter = square.perimeter();
        System.out.println(perimeter);
        int area = square.area();
        System.out.println(area);
    }
}
```

Solution 2.g)

The code could be similar to the following:

```
public class Square {
    public final int SIDES_NUMBER = 4;
    public int side;

    public Square(int l) {
        side = l;
    }

    public int perimeter() {
        int perimeter = side * SIDES_NUMBER;
        return perimeter;
    }

    public int area() {
        int area = side * side;
        return area;
    }
}
```

37

Solution 2.h)

The Rectangle class code should look like the following:

```
public class Rectangle {
    public final int NUMBER_OF_EQUAL_SIDES = 2;
    public int base;
    public int height;

    public Rectangle(int b, int h) {
        base = b;
        height = h;
    }

    public int perimeter() {
        int perimeter = (base + height ) * NUMBER_OF_EQUAL_SIDES;
        return perimeter;
    }

    public int area() {
        int area = base * height;
        return area;
    }
}
```

Solution 2.i)

The RectangleTest class code should look like the following:

```
public class RectangleTest {
    public static void main(String args[]) {
        Rectangle rectangle1 = new Rectangle(5,6);
        Rectangle rectangle2 = new Rectangle(8,9);
        System.out.println("Perimeter of rectangle 1 = "
            + rectangle1.perimeter());
        System.out.println("Area of rectangle 1 = "
            + rectangle1.area());
        System.out.println("Perimeter of rectangle 2 = "
            + rectangle2.perimeter());
        System.out.println("Area of rectangle 2 = " + rectangle2.area());
    }
}
```

Solution 2.j)

The code of the Square class could be the following (in bold the new method requested):

```
public class Square {
    public final int SIDES_NUMBER = 4;
```

```
    public int side;

    public Square(int l) {
        side = l;
    }

    public int  perimeter() {
//        int perimeter = side * 4;
        int perimeter = side * SIDES_NUMBER;
        return perimeter;
    }

    public int area() {
        int area = side * side;
        return area;
    }

    public void printDetails(){
        System.out.println("This square, has side = " + side + ", perimeter = "
          + perimeter()+ ", area = " + area());
    }
}
```

The code of the Rectangle class could be the following (in bold the new method requested):

```
public class Rectangle {
    public final int NUMBER_OF_EQUAL_SIDES = 2;
    public int base;
    public int height;

    public Rectangle(int b, int h) {
        base = b;
        height = h;
    }

    public int  perimeter() {
        int perimeter = (base + height) * NUMBER_OF_EQUAL_SIDES;
        return perimeter;
    }

    public int area() {
        int area = base * height;
        return area;
    }

    public void printDetails(){
        System.out.println("This rectangle, has base = " + base + ", height = "
          + height + ", perimeter = " + perimeter()+ ", area = " + area());
    }
}
```

The code of the SquareTest class could be the following (note how the code is simpler, shorter and more readable):

```
public class SquareTest {
    public static void main(String args[]) {
        Square square = new Square(5);
        square.printDetails();
    }
}
```

Same goes for the TestRettangolo class, whose code could be the following:

```
public class RectangleTest {
    public static void main(String args[]) {
        Rectangle rectangle1 = new Rectangle(5,6);
        Rectangle rectangle2 = new Rectangle(8,9);
        rectangle1.printDetails();
        rectangle2.printDetails();
    }
}
```

Solution 2.k)

The code of the Square class could be the following (in bold the modified code):

```
public class Square {

    public final int SIDES_NUMBER = 4;
    public int side;

    public Square(int l) {
        side = l;
    }

    public int  perimeter() {
//        int perimeter = side * 4;
        int perimeter = side * SIDES_NUMBER;
        return perimeter;
    }

    public int area() {
        int area = side * side;
        return area;
    }

    public void printDetails(){
        System.out.println(getDetails());
    }
```

```
    public String getDetails(){
        return "This square, has side = " + side + ", perimeter = " +
            perimeter()+ ", area = " + area();
    }
}
```

With the same criterion we could modify the Rectangle class with the following code (in bold the modified code):

```
public class Rectangle {
    public final int NUMBER_OF_EQUAL_SIDES = 2;
    public int base;
    public int height;

    public Rectangle(int b, int h) {
        base = b;
        height = h;
    }

    public int  perimeter() {
        int perimeter = (base + height) * NUMBER_OF_EQUAL_SIDES;
        return perimeter;
    }

    public int area() {
        int area = base * height;
        return area;
    }

    public void printDetails(){
        System.out.println(getDetails());
    }

    public String getDetails(){
        return "This rectangle, has base = " + base + ", height = " +
            height + ", perimeter = " + perimeter()+ ", area = " + area();
    }
}
```

The code of the QuadrilateralsTest class could be the following:

```
public class QuadrilateralsTest {
    public static void main(String args[]) {
        Square square = new Square(5);
        square.printDetails();
        Rectangle rectangle = new Rectangle(5,6);
        rectangle.printDetails();
    }
}
```

41

Solution 2.l)

The Nation class code should look like the following:

```java
public class Nation {
    public String name;
    public String capital;
    public int population;

    public Nation (String n, String c, int p) {
        name = n;
        capital = c;
        population = p;
    }
}
```

This abstraction, although very generic, seems correct. The only defined constructor implies that we have to specify three input parameters for every instance, so that they must be considered to be mandatory:

```java
Nation italy = new Nation("Italy", "Rome", "60000000");
```

> **You didn't have to create a class with the same variables, the important thing is to have a correct abstraction.**

Solution 2.m)

It is possible that someone has succeeded in creating methods within this class. In the previous exercise, an abstraction of the Nation class was requested only in a generic way, without specifying the context or program in which this class will have a role. This is why it is difficult for us to create methods, since we are currently ignoring the program in which Nation will be used. We could use this class in a program that preserves the physical data of the nations, but we could also use it in a video game that simulates the famous board game Risiko. The methods (but also the instance variables) to be defined, could drastically change from context to context. In the first case we would define instance variables like rivers, lakes, mountains, surfaces, etc., and methods like produce(), export(), import(). In the second case we could define the boundaries variable, and the method defend().

In conclusion, we have made the definition of the Nation class extremely generic, precisely because we had no constraints to exploit.

Solution 2.n)

The answer is the number 1, which means the code can be compiled without errors. Don't be misled by the names of the variables (see other possible answers) that could lead to confusion. Also, the order in which the modifiers are specified is not a problem.

Solution 2.o)

The mistake is that there is no ; next to the `toString()` method statement:

```
return "Exercise2O"
```

which should be corrected this way:

```
return "Exercise2O";
```

The other methods are all correct.

Solution 2.p)

The error is that a local variable cannot be declared `public` within a method. In fact, `public` defines the visibility outside the class of an instance variable, not outside a method.

Note that we have named `argz` the parameter of the `main()` method, instead of the standard `args`. This is legal because it is only a parameter name.

Solution 2.q)

The first error is that braces are missing for the `args` parameter of the `main()` method. The second is that there is no ; next to the only statement in the `main()` method. The third is due to an extra closing brace. Once these errors are corrected, the class compiles and can also be executed since it contains a `main()` method. The correct class will be the following:

```java
public class Exercise2Q_OK {
    public static void main(String args) {
        System.out.println("Quelo");
    }
}
```

Solution 2.r)

The program will not compile because of the statement:

```
System.out.println("Exercise 2.r");
```

43

which does not belong to any method code block. But we have seen that within a class only variables and methods are defined, not statements. By eliminating that statement, the program:

```java
public class Exercise2R_OK {
    public int var1;
    public int var2;

    public Exercise2R_OK() {

    }

    public Exercise2R_OK(int a , int b) {
        var1 = b;
        var2 = a;
    }

    public static void main(String args[]) {
        Exercise2R_OK exercise2R = new Exercise2R_OK(4,7);
        System.out.println(exercise2R.var1);
        System.out.println(exercise2R.var2);
    }
}
```

will compile and will print at runtime:

```
7
4
```

Solution 2.s)

The correct answer is 3. It would also be possible to compile only the Exercise2S class, since using the Course class, it will oblige the compiler to compile the latter as well. So, we can also execute this sequence

```
javac Exercise2S.java
java Exercise2S
```

Solution 2.t)

The correct answer is 3.

Solution 2.u)

The correct answer is 1. In fact, the local variable c has not been initialized and will cause the following error:

```
Exercise2U.java:6: error: variable c might not have been initialized
        System.out.println(a+b+c+d);
                                ^
1 error
```

As stated in this chapter, the instance variable has nothing to do with the local variable with the same name. In any case, once initialized to 3:

```java
public class Exercise2u_OK {
    int c = 3;
    public static void main(String args[]) {
        int a = 1;
        int b = 2, c = 3, d = 4;
        System.out.println(a+b+c+d);
    }
}
```

will print:

```
10
```

Solution 2.v)

The code for the Exercise2V class, could be the following:

```java
public class Exercise2V {
    public int sum2Int(int a, int b) {
        return a+b;
    }

    public int sum5Int(int a, int b, int c, int d, int e) {
        return a+b+c+d+e;
    }


    public int sum10Int(int a, int b, int c, int d, int e,
                        int f, int g, int h, int i, int l) {
        return a+b+c+d+e+f+g+h+i+l;
    }

    //Just for test
    public static void main(String args[]) {
        Exercise2V ex = new Exercise2V();
        System.out.println(ex.sum2Int(1,1));
        System.out.println(ex.sum5Int(1,1,1,1,1));
        System.out.println(ex.sum10Int(1,1,1,1,1,1,1,1,1,1));
    }
}
```

It would not be entirely correct to use a varargs, since it would allow us to do many other operations that are not required.

Soluzion1e 2.w)

The code of the `Contact` class could be the following:

```java
public class Contact {

    public String name;

    public String address;

    public String phoneNumber;

    public Contact(String nam, String num) {
        name = nam;
        phoneNumber = num;
    }

    public Contact(String nam, String add, String num) {
        name = nam;
        address = add;
        phoneNumber = num;
    }
}
```

Note that we have decided to define two constructors, one that takes the values of the three variables as input, the other that does without the address. We have not introduced other constructors since we consider it useless for a contact to be created without specifying at least a name and a phone number.

Soluzione 2.x)

The code of the `PrintContacts` class could be the following:

```java
public class PrintContacts {
    public static void main(String args[]) {
        System.out.println("Contacts List");
        System.out.println();
        Contact contact1 = new Contact("Claudio De Sio Cesari",
            "13, Java Street", "131313131313");
        Contact contact2 = new Contact("Stevie Wonder", "10, Music Avenue",
            "1010101010");
        Contact contact3 = new Contact("Gennaro Capuozzo",
            "1, Four Days of Naples Square" ,"1111111111");
        System.out.println(contact1.name);
        System.out.println(contact1.address);
        System.out.println(contact1.phoneNumber);
```

```
        System.out.println();
        System.out.println(contact2.name);
        System.out.println(contact2.address);
        System.out.println(contact2.phoneNumber);
        System.out.println();
        System.out.println(contact3.name);
        System.out.println(contact3.address);
        System.out.println(contact3.phoneNumber);
    }
}
```

However, the code is rather verbose.

As recommended, we add a `printDetails()` method in the `Contact` class (see code in bold):

```
public class Contact {
    public String name;

    public String address;

    public String phoneNumber;

    public Contact(String nam, String num) {
        name = nam;
        phoneNumber = num;
    }

    public Contact(String nam, String add, String num) {
        name = nam;
        address = add;
        phoneNumber = num;
    }

    public void printDetails() {
        System.out.println(name);
        System.out.println(address);
        System.out.println(phoneNumber);
        System.out.println();
    }
}
```

We can rewrite the `PrintContacts` class more easily (changes in bold):

```
public class PrintContactsV2 {
    public static void main(String args[]) {
        System.out.println("Contacts List");
        System.out.println();
        Contact contact1 = new Contact("Claudio De Sio Cesari",
            "13, Java Street", "131313131313");
        Contact contact2 = new Contact("Stevie Wonder", "10, Music Avenue",
            "1010101010");
```

```
            Contact contact3 = new Contact("Gennaro Capuozzo",
                "1, Four Days of Naples Square", "1111111111");
            contact1.printDetails();
            contact2.printDetails();
            contact3.printDetails();
        }
    }
```

Solution 2.y)

The code of the PhoneBook class could be the following:

```
public class PhoneBook {
    public Contact contact1;
    public Contact contact2;
    public Contact contact3;
    public PhoneBook() {
        contact1 = new Contact("Claudio De Sio Cesari",
            "13, Java Street", "131313131313");
        contact2 = new Contact("Stevie Wonder",
            "10, Music Avenue", "1010101010");
        contact3 = new Contact("Gennaro Capuozzo",
            "1, Four Days of Naples Square", "1111111111");
    }
}
```

As a result, the PrintContacts class can be changed as follows (changes in bold):

```
public class PrintContacts {
    public static void main(String args[]) {
        System.out.println("Contacts List");
        System.out.println();
        PhoneBook phoneBook = new PhoneBook();
        phoneBook.contact1.printDetails();
        phoneBook.contact2.printDetails();
        phoneBook.contact3.printDetails();
    }
}
```

Solution 2.z)

One solution could be the coding of the following classes:

```
public class City {
    public String name;

    public City(String n) {
        name = n;
    }
}
```

48

```
public class Nation {
    public String name;
    public City capital;
    public int population;

    public Nation(String n, City c, int p) {
        name = n;
        capital = c;
        population = p;
    }
}

public class Exercise2z {
    public static void main(String args[]) {
        City city = new City("Rome");
        Nation nation = new Nation("Italy", city, 60000000);
        System.out.println(nation.name + " has " + city.name
                            + " as its capital");
    }
}
```

Chapter 3
Exercises

Coding Style, Data Types and Arrays

Below is a series of exercises to practice what was learned in Chapter 3. We remember once again, that many exercises are preparatory to the following ones, so it is not worth skipping some, and we recommend you to consult at least the solutions before going ahead.

Exercise 3.a)

Write a simple program that performs the following arithmetic operations correctly, carefully choosing the data types to be used to store their results.

1. A division (use the / symbol) between two integers a = 5, and b = 3. Store the result in a variable r1, choosing the data type appropriately.

2. A multiplication (use the * symbol) between a char c = 'a', and a short s = 5000. Store the result in a variable r2, choosing the type of data appropriately.

3. A sum (use the + symbol) between an int i = 6 and a float f = 3.14F. Store the result in a variable r3, choosing the data type appropriately.

4. A subtraction (use the – symbol) between r1, r2 and r3. Store the result in a variable r4, choosing the data type appropriately.

Verify the correctness of the operations by printing the partial results and the final result. Keep

in mind the automatic promotion in expressions, and use the casting appropriately. Write all the code in a class with a `main()` method.

Exercise 3.b)

Write a program with the following requirements.

■ Implement a `Person` class that declares the variables `name`, `surname`, `age`. Also create a `details()` method that returns information about the person object with a string. Remember to use the conventions and rules described in this chapter.

■ Implement a `Main` class that, in the `main()` method, presents two objects called `person1` and `person2` of the `Person` class, initializing the relative fields for each of them through the dot operator.

■ Declare a third reference (`person3`) that points to one of the objects already instantiated. Check that actually `person3` points to the desired object, printing the `person3` fields always using the dot operator.

■ Adequately comment the code and use the **javadoc** tool to produce the related documentation.

> All the rules and conventions described in this chapter are used in the standard Java documentation. Just observe that `String` starts with a capital letter, being a class. Obviously, `System` is also a class.

Exercise 3.c) Arrays, True or False:

1. An array is an object and can therefore be declared, instantiated and initialized.

2. A two-dimensional array is an array whose elements are other arrays.

3. The `length` method returns the number of elements of an array.

4. An array is not resizable.

5. An array is heterogeneous by default.

6. An array of integers can contain `byte` types as elements, that is, the following lines of code do not produce compilation errors:

```
int arr [] = new int[2];
byte a = 1, b=2;
arr [0] = a;
arr [1] = b;
```

7. An array of integers can contain `char` types as elements, that is, the following lines of code do not produce compilation errors:

```
char a = 'a', b = 'b';
int arr [] = {a,b};
```

8. An array of strings can contain `char` types as elements, that is, the following lines of code do not produce compilation errors:

```
String arr [] = {'a' , 'b'};
```

9. An array of strings is a two-dimensional array, because strings are nothing more than arrays of characters. For example:

```
String arr [] = {"a" , "b"};
is a two-dimensional array.
```

10. Given the following two-dimensional array:

```
int arr [][]= {
    {1, 2, 3},
    {1,2},
    {1,2,3,4,5}
};
```

it will turn out that:

```
arr.length = 3;
arr[0].length = 3;
arr[1].length = 2;
arr[2].length = 5;
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 1;
arr[1][1] = 2;
arr[1][2] = 3;
arr[2][0] = 1;
arr[2][1] = 2;
arr[2][2] = 3;
arr[2][3] = 4;
arr[2][4] = 5;
```

53

Exercise 3.d)

Create a `PrintMyName` class with a `main()` method, which prints your name using an array of characters.

Exercise 3.e)

Create a class `Result` that declares a single instance variable of type `float` named `result`. Add any useful methods and constructors. Create a `ChangeResult` class that declares a public method of name `changeResult()` which takes an object of type `Result` and changes its local variable `result` by adding to it another value.

Create a class with a `main()` method named `ResultTest` that prints the `result` variable of an object of type `Result`, before and after this object is passed as input to the method `changeResult()` of an object of type `ChangeResult`.

Exercise 3.f)

After having done the previous exercise, add a method called `changeResult()` to the class `ChangeResult` which takes a `float` parameter as input and changes the `result` variable.

Then create an equivalent `ResultFloatTest` class, which performs the same operations as the `ResultTest` class realized in the previous exercise.

Exercise 3.g)

Create a class named `ArgsTest` with a `main()` method that prints the variable `args[0]`. Then test it by passing various arguments by command line (see section 3.6.5).

Exercise 3.h)

The following class declares several string identifiers:

```
public class Exercise3H {
    public String Break,
    String,
    character,
    bit,
    continues,
    exports,
    Class,
    imports,
    _AAA_;
```

```
    _@_,
    _;
}
```

Are they all valid? Once the invalid ones have been identified, use the comments appropriately to exclude them from the compilation.

Exercise 3.i)

Given the following class:

```
package com.claudiodesio.java.exercises;

public class Exercise3I {
    public final String languageName = "Java";
    public int integer;
    public void printString(){
        System.out.println(languageName);
    }
}
```

Is there any convention not respected for identifiers? If yes, which changes should be done? If the naming conventions are not respected, the code does not compile?

Exercise 3.j)

Considering the Exercise 1.y, create a class called `PrintVoidRowClass` that prints the following ouput:

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

Exercise 3.k)

Given the following class:

```
public class Exercise3K {
    public static void main(String args[]) {
        char d = (char)100;
        float \u0066 = (float)d*1_000_000_000;
        System.out.println((long)f);
    }
}
```

Keeping in mind that the letter f, is encoded in Unicode by the hexadecimal number 66, which of the following outputs will be produces once executed?

1. None, we will have a compile error for a syntax error on the second line of the `main()` method.

2. An unknown number.

3. `100000000000`

4. `100_000_000_000`

5. `100.000.000.000`

6. `100,000,000,000`

7. `66000000000`

8. `0.0`

9. An unpredictable Unicode character.

10. The code compiles but at runtime we will have an exception in the first line of the `main()` method because it is not possible to cast from `int` to `char`.

11. The code compiles but at runtime we will have an exception in the second line of the `main()` method because it is not possible to cast from `int` to `float`.

12. The code compiles but at runtime we will have an exception i the third line of the `main()` method because it is not possible to cast from `float` to `long`.

13. The code does not compile for other reasons.

14. The code throws a runtime exception for other reasons.

Exercise 3.l)

Given the following class:

```java
public class Exercise3L {
    public static void main(String args[]) {
        bit i1 = 8;
        short i2 = -1024;
        integer i3 = 638;
        long i5 = 888_666_777;
        float i6 = 0;
        double i7 = 0x11B;
        System.out.println(i7);
    }
}
```

Are the variables inside the `main()` method all declared correctly? And are all the assigned values within the range of representation of the respective types?

Exercise 3.m)

Given the following class:

```java
public class Exercise3M {
    public static void main(String args[]) {
        boolean b = true;
        char c = 'I';
        System.out.println(b);
        System.out.println(c+1);
    }
}
```

Bearing in mind that the letter I is encoded by the number 73, which of the following outputs will be produced once executed?

1. We will have an error in compilation.

2. `true` to the first line and 74 to the second.

3. `true` to the first line and L to the second.

4. `true` to the first line and J to the second.

5. `0` to the first line and 74 to the second.

6. `0` to the first line and J to the second.

7. `0` to the first line and L to the second,

Exercise 3.n)

Given the following class:

```java
public class Exercise3N {
    public static void main(String args[]) {
        String s = "Jav";
        char c = "a";
        System.out.println(s+c+1);
    }
}
```

which of the following outputs will be produced once executed?

1. We will have an error in compilation.

2. Java

3. Java1

4. Javb

Exercise 3.o)

Given the following class:

```
package parking;

public class Car {
    public String type;

    public Car(String t){
        type = t;
    }
}
```

For the following class to compile:

```
package workers;
//Insert code here

public class Driver {
    public void drive(Car car) {
        System.out.println("I'm driving a " + car.type + " car" );
    }
}
```

You must insert a line of code. Which between the following lines would allow the `Driver` class to be compiled (choose all that applies)?

1. import parking.Car;

2. import parking.*;

3. import parking.workers.*;

4. import parking.Car.*;

5. import parking.*.Car;

6. import workers.parking.Car;

Exercise 3.p)

Considering the `Car` and `Driver` classes of the previous exercise, which piece of code must be added to the following class:

```
public class Exercise3P {

//Insert code here

    public static void main(String args[]) {
        Car car = new Car("Toyota Yaris");
        Driver driver = new Driver();
        driver.drive(car);
    }
}
```

Choose only one of the following options:

1. `import parking.*;`

2. `import workers.*;`

3. `import parking.Car; e import workers.Driver;`

4. No code needs to be added. The code can already be compiled.

Exercise 3.q)

The following statement is correct?

■■ When we pass a reference of an object to a method as input, we are sure that once the method has been executed, our reference will always point to the same object it was pointing to before the method was executed. This does not mean that the internal structure of the object cannot be modified within the method. In fact, the local parameter of the method will have the same address as the reference passed, and can therefore work on the same object.

Exercise 3.r)

Write a program that takes an argument as input (variable `args` of the `main()` method) and store it as a third element of an array of local strings named `array`.

> **This program will only work if at least one argument is passed. Any parameters passed from the command line beyond the first one will be ignored by the program.**

Exercise 3.s)

Given the following class:

```
public class Car {
    public String type;

    public Car(String t){
        type = t;
    }
}
```

Create a Boat class that abstracts the concept of a boat that loads cars. This class must define a loadCar() method to which a Car object will be passed. Each Car object will be stored in an array instance variable called carArray, as first element. Also create a test class that you can call Exercise3S.

Exercise 3.t)

Create an Exercise3T class that must be launched by passing a command line argument representing an integer, as follows:

> **With EJE it is possible to pass command line arguments with the keyboard shortcut** Shift - F9**, or by clicking on the menu** execute with args.

You can replace the number 9 with any other integer number.
The program will have to:

1. use args[0], which contains an integer, to create an array of integers of the same size specified by the argument;

2. print a sentence that will confirm the creation of the array, printing its size.

Since the parameters are stored in the args string array elements, we need to convert the parameter passed as input from string to integer. Search the library for the parseInt() method of the Integer class. Read the documentation, understand how it works and use it in the program.

> **In case you are not able to do some task, search Google for help, the web is full of solutions.**

Exercise 3.u)

Create a program that:

1. create an array of character types, containing all the letters of the alphabet;

2. use a (static) method of the `java.util.Arrays` class to print its contents as a string. Look for the appropriate method in the official documentation.

> **In case you are not able to do some task, search Google for help, the web is full of solutions.**

Exercise 3.v)

Create a class that prints integer random numbers.

> **Hint: there is a method of a class in the Java library that does exactly this. To find it, search for the word "random" in the documentation.**

Exercise 3.w)

Given the following class:

```
public class Exercise3W {
    public static void main(String args[]) {
        var var = "var";
        var a = " ";
        var b = " = ";
        var c = 8;
        var d = ";";
        var e = c + d;
        System.out.println(var + a + "i" + b + c +d);
    }
}
```

Is it possible to compile this application?

1. No, we will have a compilation error due to a syntax error on the first line of the `main()` method.

2. No, we will have a compilation error due to a syntax error on the second line of the `main()` method.

3. No, we will have a compilation error due to a syntax error on the third line of the `main()` method.

4. No, we will have a compilation error due to a syntax error on the fourth line of the `main()` method.

5. No, we will have a compile error for a syntax error on the fifth line of the `main()` method.

6. No, we will have a compile error for a syntax error on the sixth line of the `main()` method.

7. No, we will have a compilation error due to a syntax error on the seventh line of the `main()` method.

8. Yes.

Exercise 3.x)

Rewrite the solution of the Exercise 2.y, using the word var, wherever it is possible to use it. Also in the PhoneBook class, replace the three instance variables `contact1`, `contact2` and `contact3` with an array, and modify the `PrintContacts` class accordingly.

Exercise 3.y)

Which of the following snippets can be compiled without errors:

1. `public class var {}`

2. `private class var {}`

3. `public var MyClass {}`

4. `var var[] = new int[8];`

5. `public var var = 1;`

Exercise 3.z)

Create a `ReportCard` class that abstracts the concept of school report. It must have the following information:

1. name, surname and class of the student;

2. a table of votes that you associate for each subject, the vote and the judgment

3. It must also declare a method that reads the report data legibly.

4. Also create an `Exercise3Z` class that prints one or more report cards.

Chapter 3
Exercise Solutions

Coding Style, Data Types and Arrays

```java
public class Exercise3A {
    public static void main (String args[]) {
        int a = 5, b = 3;
        double r1 = (double)a/b;
        System.out.println("r1 = " + r1);
        char c = 'a';
        short s = 5000;
        int r2 = c*s;
        System.out.println("r2 = " + r2);
        int i = 6;
        float f = 3.14F;
        float r3 = i + f;
        System.out.println("r3 = " + r3);
        double r4 = r1 - r2 - r3;
        System.out.println("r4 = " + r4);
    }
}
```

Solution 3.b)

```java
public class Person {
    public String name;
    public String surname;
    public int age;
    public String details() {
        return name + " " + surname + " years " + age;
    }
}
public class Main {
    public static void main (String args []) {
        Person person1 = new Person();
        Person person2 = new Person();
        person1.name = "Alessandro";
        person1.surname = "Scarlatti";
        person1.age = 30;
        System.out.println("person1 "+person1.details());
        person2.name = "Antonio";
        person2.surname = "Vivaldi";
        person2.age = 40;
        System.out.println("person2 "+person2.details());
        Person person3 = person1;
        System.out.println("person3 "+person3.details());
    }
}
```

Solution 3.c) Array, True or False:

1. **True**.

2. **True**.

3. **False**, the `length` variable returns the number of elements in an array.

4. **True**.

5. **False**.

6. **True**, a `byte` (which takes only 8 bits) can be stored in an `int` variable (which takes 32 bits).

7. **True**, a `char` (which takes 16 bits) can be stored in an `int` variable (which takes 32 bits).

8. **False**, a `char` is a primitive data type and `String` is a class. The two types of data are not compatible.

9. **False**, in Java the string is an object instantiated by the String class and not an array of characters (even if internally uses an array of characters).

10. **False**, all the statements are correct except arr[1][2] = 3; because this element does not exist.

Solution 3.d)

The code should be similar to the following:

```
public class PrintMyName {
    public static void main(String args[]) {
        char [] name = {'C', 'l', 'a', 'u', 'd', 'i', 'o'};
        System.out.println(name);
    }
}
```

Solution 3.e)

The code of the Result class could be the following:

```
public class Result {
    public float result;

    public Result (float res) {
        result = res;
    }

    public void print() {
        System.out.println(result);
    }
}
```

Note that we have created a constructor and a method to facilitate printing.
The code of the ChangeResult class could be the following:

```
public class ChangeResult {
    public void changeResult(Result result) {
        result.result += 1;
    }
}
```

The code of the ResultTest class could be the following:

```
public class ResultTest {
    public static void main(String args[]) {
        Result result = new Result(5.0F);
```

67

```
        result.print();
        ChangeResult cr = new ChangeResult();
        cr.changeResult(result);
        result.print();
    }
}
```

The output of the previous code will be:

```
5.0
6.0
```

Solution 3.f)

The code of the `ChangeResult` class should change as follows:

```
public class ChangeResult {
    public void changeResult(Result result) {
        result.result += 1;
    }
    public float changeResult(float result) {
        result += 1;
        return result;
    }
}
```

Note that this time the method must return the new value of the variable since it is a primitive type variable (see section 3.3).
The code of the `ResultFloatTest` class could be the following:

```
public class ResultFloatTest {
    public static void main(String args[]) {
        float result = 5.0F;
        System.out.println(result);
        ChangeResult cr = new ChangeResult();
        result = cr.changeResult(result);
        System.out.println(result);
    }
}
```

Note that we had to reassign the value of the `result` variable after the computation of the `changeResult()` method.

Solution 3.g)

The code should be similar to:

```
public class TestArgs {
    public static void main(String args[]) {
        System.out.println(args[0]);
    }
}
```

Note that if you don't specify an argument when you launch the application you will get an exception at runtime:

```
java TestArgs
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at TestArgs.main(TestArgs.java:3)
```

The exceptions are discussed in Chapter 9.

Solution 3.h)

Only the last two identifiers are not valid, in fact:

1. The Break identifier is different from break (all keywords do not have uppercase letters).

2. The String identifier coincides with the name of the String class, but not being a keyword, you can use it as an identifier. Nevertheless, it is a bad practice.

3. The character identifier is not a keyword (there is char instead).

4. The bit identifier is not a keyword (there is byte instead).

5. The continues identifier is not a keyword (there is continue instead).

6. The exports identifier is a restricted word, and would be unusable within the declaration of a module, but in this context, it does not create problems.

7. The Class identifier is not a keyword (there is class instead).

8. The imports identifier is not a keyword (there is import instead).

9. The _AAA_ identifier is not a keyword.

10. The identifier _@_ is not legal because we can't use the @ symbol as part of identifiers.

11. The identifier _ is a reserved word starting from Java 9.

You can comment out the invalid identifiers in the following way:

```
public class Exercise3H {
    public String Break,
```

```
        String,
        character,
        bit,
        continues,
        exports,
        Class,
        imports,
        _AAA_/*,
        _@_,
        _*/;
    }
```

We have used a multi-line comment to comment out only on what should be commented. Clearly this approach does not favour to the readability of the code. The most appropriate way to comment out our code requires the replacement of the "," symbol with the ";" symbol immediately after the declaration of the identifier _AAA_, and the use of the single-line comments:

```
public class Exercise3H {
    public String Break,
        String,
        character,
        bit,
        continues,
        exports,
        Class,
        imports,
        _AAA_;
//      _@_,
//      _*/;
    }
```

Solution 3.i)

The only convention not used correctly concerns the constant (we remind you that the conventions do not affect the compilation of the code).
The code should be corrected as follows:

```
package com.claudiodesio.java.exercises;

public class ExerciseSolution3I {
    public final String LANGUAGE_NAME = "Java";
    public int integer;
    public void printString(){
        System.out.println(LANGUAGE_NAME);
    }
}
```

70

Solution 3.j)

The problem arises for printing double quotes defined within the `println()` method. The solution is to escape the double quotes (in bold):

```java
public class PrintVoidRowClass {
    public static void main(String args[]) {
        System.out.println("public class VoidRow {");
        System.out.println("    public static void main(String args[]) {");
        System.out.println("        System.out.println(\"\");");
        System.out.println("    }");
        System.out.println("}");
    }
}
```

Solution 3.k)

The code compiles and runs without errors, and prints:

```
99999997952
```

that is, an unknown number. Indeed, the float types, due to the limitations of the IEEE-754 standard, when exceed the 9 decimal places can use approximate numbers (see section 3.3.2.1). The correct answer is therefore to number 2.

Solution 3.l)

The `bit` and `integer` types do not exist (if anything, there are `byte` and `int`). All declared values are compatible with the respective representation ranges, including the value `0x11B` which is `283.0` for the decimal system and which, being stored in a `double`, is absolutely compatible.

Solution 3.m)

The right answer is the number 2. In fact, a boolean literal will be printed exactly as its literal value (`true` in this case). Instead `c + 1` is promoted to an integer, and from 73 it becomes 74. To be able to print the relative character value (`J`) we should cast the whole operation in this way:

```java
System.out.println((char)(c+1));
```

Solution 3.n)

We will have a compile-time error because the value assigned to the character `c`, is a string (note the double quotes instead of the single quotes).

Solution 3.o)

The correct answers are 1 and 2, all the others are incorrect contain syntax errors.

Solution 3.p)

The correct answer is 3 because both `Car` and `Driver` classes were used in the code.

Solution 3.q)

Yes, the reasoning is correct.

Solution 3.r)

The code should be similar to:

```
public class Exercise3R {
    public static void main(String args[]) {
        String[] array = new String[5];
        array[2] = args[0];
    }
}
```

Solution 3.s)

The required code should be similar to this:

```
public class Boat {
    int index = 0;
    public Car[] carArray;

    public Boat () {
        carArray = new Car[100];
    }

    public void loadCar(Car car) {
        carArray[index] = car;
        System.out.println("Car: "+ car.type +" loaded");
        index++;
    }
}
```

Where we used an index to keep track of the positions already occupied on the boat. This is increased each time a car is loaded, and then used to load the next one.
The following test class satisfies the request:

72

```
public class Exercise3S {
    public static void main(String args[]) {
        Boat boat = new Boat();
        Car car1 = new Car("Renault");
        Car car2 = new Car("Volkswagen");
        Car car3 = new Car("Nissan");
        boat.loadCar(car1);
        boat.loadCar(car2);
        boat.loadCar(car3);
    }
}
```

Solution 3.t)

The required code should be similar to the following:

```
public class Exercise3T {
    public static void main(String args[]) {
        int arrayDimension = Integer.parseInt(args[0]);
        int [] array = new int[arrayDimension];
        System.out.println("The array has dimension " + array.length);
    }
}
```

Note that the parseInt() method is static (argument not yet addressed) and can be used with the syntax: ClassName.parseInt(), but it is also possible to instantiate an object and invoke it as it was an ordinary method (but it is useless to instantiate the object).

Solution 3.u)

The required code should be similar to the following:

```
import java.util.Arrays;

public class Exercise3U {
    public static void main(String args[]) {
        char[] array = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
        'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
        't', 'u', 'v', 'w', 'x', 'y', 'z'};
        System.out.println(Arrays.toString(array));
    }
}
```

The toString() method of the Arrays class was the required method. To use it you need to import the Arrays class.

Solution 3.v)

The required code should be similar to the following:

```
import java.util.Random;

public class Exercise3V {
    public static void main(String args[]) {
        Random random = new Random();
        System.out.println(random.nextInt());
    }
}
```

Solution 3.w)

The Exercise3W class will be compiled correctly and when executed will print:

```
var i = 8;
```

Solution 3.x)

We can replace instance variables of the PhoneBook class with an array, using the following code:

```
public class PhoneBook {
  public Contact [] contacts;
  public PhoneBook () {
    contacts = new Contact[]{
    new Contact("Claudio De Sio Cesari", " 13, Java Street ", "131313131313"),
      new Contact("Stevie Wonder", "10, Music Avenue", "1010101010"),
      new Contact("Gennaro Capuozzo", " 1, "Four Days of Naples Square ",
        "1111111111")};
    }
}
```

but we can also use the ordinary syntax of the array:

```
public class PhoneBook {
    public Contact [] contacts;
    public PhoneBook () {
        contacts = new Contact[3];
        contacts[0] = new Contact("Claudio De Sio Cesari", "13, Java Street",
            "131313131313");
        contacts[1] = new Contact("Stevie Wonder", "10, Music Avenue",
            "1010101010");
        contacts[2] = new Contact("Gennaro Capuozzo",
            "1, Four Days of Naples Square", "1111111111");
    }
}
```

74

As for the introduction of the word var, in the Contact and PhoneBook classes, since no local variable is defined, it is not possible to use the word var, so these classes should not be modified. The PrintContacts class instead, can be modified in this way (in bold the change):

```java
public class PrintContacts {
    public static void main(String args[]) {
        System.out.println("Contacts List");
        System.out.println();
        var phoneBook = new PhoneBook();
        phoneBook.contacts[0].printDetails();
        phoneBook.contacts[1].printDetails();
        phoneBook.contacts[2].printDetails();
    }
}
```

Solution 3.y)

No snippets are correct. The numbers 1 and 2 show classes with the var identifier, but it is not possible to use the word var as an identifier for any type (see section 3.7.2). Furthermore, in the case of the snippet number 2, it is not even possible to use the private keyword to define a class (we will see it better in the next chapters). In snippet 3, there is a syntax that tries to define a class using the var keyword instead of the class keyword, but this is not the function for which the word var was created. In the fourth snippet we try to use the word var as an identifier of an array, but again in section 3.7.2, it is clearly specified that this is illegal. Finally, in snippet number 5, we can infer from the presence of the public modifier, that it is a definition of an instance variable and not a local variable, and therefore the use of the word var is not allowed.

Solution 3.z)

We decide to create an abstraction for the Student class:

```java
public class Student {
    public String name;
    public String surname;
    public String schoolClass;

    public Student(String nam, String sur, String sc) {
        name = nam;
        surname = sur;
        schoolClass = sc;
    }

    public String toString() {
        return "Student: "+ name +" "+ surname +"\nClass "+ schoolClass;
    }
}
```

75

We have declared the essential information required, and we have created a constructor to set this information. We also created a toString() method that returns a descriptive string of the object.

> **We will see later that this method will be used very often in Java programming, because it is already present in every class.**

Then we create a ReportCard class that abstracts the concept of a table of votes:

```java
import java.util.Arrays;

public class ReportCard {
    public Student student;
    public String[][] tableOfVotes;

    public ReportCard (Student stu, String [][] tab){
        student = stu;
        tableOfVotes = tab;
    }

    public void printReportCard() {
        System.out.println(student.toString());
        System.out.println(Arrays.toString(tableOfVotes[0]));
        System.out.println(Arrays.toString(tableOfVotes[1]));
        System.out.println(Arrays.toString(tableOfVotes[2]));
        System.out.println(Arrays.toString(tableOfVotes[3]));
        System.out.println(Arrays.toString(tableOfVotes[4]));
        System.out.println(Arrays.toString(tableOfVotes[5]));
        System.out.println(Arrays.toString(tableOfVotes[6]));
    }
}
```

Note that this class declares a student object and a two-dimensional array tableOfVotes, both to be set when the object is instantiated with the provided constructor. It also declares the method printingReportCard() which uses the static method toString() of the class java.util.Arrays to format the content of each "row" of the two-dimensional array tableOfVotes.

Finally, with the following class, let's print two report cards:

```java
public class Exercise3Z {
    public static void main(String args[]) {
        Student student1 = new Student("Giovanni","Battista","5A");
        String [][] tabellaVoti1 = {
            {"English","7","Does not engage too much."},
            {"Maths","9","He is very fit for this subject."} ,
```

```
                    {"History","7","He could do more."} ,
                    {"Geography","8","Passionate."} ,
                    {"French","9", "Able to support dialogues."},
                    {"Physical Education and Sports","6", "Vote of encouragement."},
                    {"Music","7", "He has passion for the subject."}
            };
            ReportCard reportCard1 = new ReportCard (student1, tabellaVoti1);

            Student student2 = new Student("Daniele","Sapore","2A");
            String [][] tabellaVoti2 = {
                    {"English","8","He shows enthusiasm for the subject."},
                    {"Maths","5","Not at all interested."} ,
                    {"History","6","Interested, but makes little effort."} ,
                    {"Geography","6","He could do more."} ,
                    {"French","8", "Excellent pronunciation."},
                    {"Physical Education and Sports","7", "A bit lazy."},
                    {"Music","9",
                     "He plays different instruments and has a great voice."}
            };

            ReportCard reportCard2 = new ReportCard (student2, tabellaVoti2);

            reportCard1.printReportCard();
            reportCard2.printReportCard();

        }
    }
```

The output will be the following:

```
Student: Giovanni Battista
Class 5A
[English, 7, Does not engage too much.]
[Maths, 9, He is very fit for this subject.]
[History, 7, He could do more.]
[Geography, 8, Passionate.]
[French, 9, Able to support dialogues.]
[Physical Education and Sports, 6, Vote of encouragement.]
[Music, 7, He has passion for the subject.]
Student: Daniele Sapore
Class 2A
[English, 8, He shows enthusiasm for the subject.]
[Maths, 5, Not at all interested.]
[History, 6, Interested, but makes little effort.]
[Geography, 6, He could do more.]
[French, 8, Excellent pronunciation.]
[Physical Education and Sports, 7, A bit lazy.]
[Music, 9, He plays different instruments and has a great voice.]
```

Chapter 4
Exercises

Operators and
Execution Flow Management

After studying the fourth chapter, we should already be able to write programs with Java. What is still missing are the concepts of Object Orientation that we will study starting from the next chapter. Meanwhile, we should be familiar with the execution flow.

Exercise 4.a)

Write a simple program consisting of a single class, which using only an infinite loop, the modulo operator, two `if` constructs, a `break` and a `continue`, print only the first five even numbers.

Exercise 4.b)

Write an application that prints the 26 characters of the alphabet with a loop.

Exercise 4.c)

Write a simple class that prints out the multiplication table.

Tip 1: Arrays are not required.

79

> **Tip 2: the** `System.out.println()` **method prints the argument passed as input to it, and then moves the cursor to the next line; in fact,** `println` **stands for "print line". There is also the** `System.out.print()` **method which instead prints only the argument passed to it.**
>
> **Tip 3: take advantage of a double nested loop.**

Exercise 4.d) Operators and Execution Flow, True or False:

1. The unary pre-increment and post-increment operators applied to a variable give the same result. In fact, if we have:

    ```
    int i = 5;
    ```
    or
    ```
    i++;
    ```
    or
    ```
    ++i;
    ```

    nothing changes; the value of i is updated to 6;

2. d += 1 is the same as d++, where d is a double.

3. If we have:

    ```
    int i = 5;
    int j = ++i;
    int k = j++;
    int h = k--;
    boolean flag = ((i != j) && ( (j <= k ) ||  (i <= h)) );
    ```

    flag will be false.

4. The instruction:

    ```
    System.out.println(1 + 2 + "3");
    ```

    will print 33.

5. The switch construct in any case can replace the if construct.

6. The ternary operator in any case can replace the if construct.

7. The for construct in any case can replace the while construct.

8. The do construct in any case can replace the while construct.

9. The switch construct in any case can replace the while construct.

10. The break and continue instructions can be used in the switch, for, while and do constructs, but not in the if construct,

Exercise 4.e)

Modify the ArgsTest created in the Exercise 3.g, so as to avoid runtime exceptions, with a construct learned in this chapter.

Exercise 4.f)

It is good practice to add the default clause in a switch construct. Can you explain why?

Exercise 4.g)

It is a good idea to add the else clause to an if construct. Can you explain why?

Exercise 4.h)

Create a class with a main() method that selects the first 10 numbers divisible by 3, and concatenate and print them with a string, so that the output of the program is:

```
Number multiple of 3 = 3
Number multiple of 3 = 6
Number multiple of 3 = 9
...
```

Use an ordinary for loop.

Exercise 4.i)

Repeat the Exercise 4.h using a while loop instead of a for loop.

Exercise 4.j)

Repeat the Exercise 4.j using a do-while loop instead of a for loop.

81

Exercise 4.k)

Create a class called EvenOrOdd that defines a method called getEvenOrOdd(), which randomly returns the string "Even" or "Odd". Also create a test class that invokes this method and prints the result.

> **Tip: see the solution of the Exercise 3.v.**

Exercise 4.l)

Using the EvenOrOdd class created in the previous exercise, create a class called HeadsOrTails that defines a method called getHeadsOrTails() that using a switch expression, returns the string "Heads" or "Tails". Also create a test class that invokes this method and prints the result.

Exercise 4.m)

Consider the following code:

```java
import java.util.Scanner;

public class InteractiveApp {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String string = "";
        System.out.println("Type something then press enter, " +
          "or type \"end\" to end the program");

        while (!(string = scanner.next()).equals("end")) {
            System.out.println("You typed " + string.toUpperCase() + "!");
        }

        System.out.println("Program ended!");
    }
}
```

This class reads keyboard input using the Scanner class of the java.util package (which we will discuss in the Chapter 14). The next() method used in the while construct (with a complex syntax that also includes the assignment to the string variable) is a *blocking method* (that is, that blocks the execution of the code waiting for user input) that reads the input from the keyboard, until the **enter** key is pressed. The program ends when you type the word "end".
Edit the previous program so that it becomes a *word moderator*, meaning that it must censor some words you type.

> **Perform the exercise only by censoring the words typed individually (not within a sentence), unless you are convinced that you are able to do it (the documentation is as always at your disposal to find methods useful to do what you want).**

Exercise 4.n)

Which of the following operators can be used with boolean variables?

1. +

2. %

3. ++

4. /=

5. &

6. =

7. !!

8. >>>

Exercise 4.o)

What is the output of the following program?

```java
public class Exercise4O {
    public static void main(String args[]) {
        int i = 99;
        if (i++ >= 100) {
            System.out.println(i+=10);
        } else {
            System.out.println(--i==99?i++:++i);
        }
    }
}
```

Choose from the following options:

1. 10

2. 101

3. 99

4. 110

What is the output of the following program?

```java
public class Exercise4P {
    public static void main(String args[]) {
        int i = 22;
        int j = i++%3;
        i = j!=0?j:i;
        switch (i) {
            case 1:
            System.out.println(8<<2);
            case 0:
            System.out.println(8>>2);
            break;
            case 2:
            System.out.println(i!=j);
            break;
            case 3:
            System.out.println(++j);
            break;
            default:
            System.out.println(i++);
            break;
        }
    }
}
```

Choose from the following options:

1. 24

2. 6 and on the next line 10

3. 10

4. true

5. false

6. 22

7. 21

8. 32 and on the next line 2

84

Exercise 4.q)

Write a program that asks the user to enter the number of days passed since his last vacation. Once this number is entered, the program will have to print how many minutes have passed since the last vacation.

Exercise 4.r)

Given the following class:

```
public class Exercise4R {

    private static int matrix[][] = {
        {1, 7, 3, 9, 5, 3},
        {6, 2, 3},
        {7, 5, 1, 4, 0},
        {1, 0, 2, 9, 6, 3, 7, 8, 4}
    };

    public static void main(String args[]) {

    }
}
```

implement the `main()` method so that it reads a number (between 0 and 9) as parameter `args[0]`, and find the position (row and column) of the first occurrence of the number specified within the two-dimensional array called `matrix`.

Exercise 4.s)

The solution of the previous exercise fails when:

1. no command line argument is specified;
2. an integer argument from the command line is specified, that is not within the range 0-9;
3. an argument from the command line is specified that is not an integer.

Add, to the solution of the Exercise 4.r, the code that manages the three cases (specifying a message with the instructions to follow for correct use).

> **The third case can be managed with what has been studied so far, but later we will study methods to simplify our code. In particular in Chapter 9 where we talk about exception handling, and in Chapter . . .**

85

. . . 14 where we talk about regular expressions, we will see that there are quite simple solutions for managing the third case.

Exercise 4.t)

Declare a `SimpleCalc` class that given two numbers, defines the methods for:

1. Summing them.

2. Subtract the second from the first.

3. Multiply.

4. Divide them.

5. Return the rest of the division.

6. Return the largest number (the maximum).

7. Return the smallest number (the minimum).

8. Return the average of the two numbers

9. Create a class that tests all methods.

Exercise 4.u)

Declare a class using the `Scanner` class, which allows the user to interact with the `SimpleCalc` class: the user must be able to write the first operand, select the operation to be performed from a list and specify the second operand. The program must return the right result.

Exercise 4.v)

Declare a `StrangeCalc` class that given an unspecified number of numbers, defines the methods for:

1. Summing them.

2. Subtract the second from the first.

3. Multiply.

4. Divide them.

5. Return the rest of the division.

6. Return the largest number (the maximum).

7. Return the smallest number (the minimum).

8. Return the average of the two numbers.

Exercise 4.w)

Declare a class that uses the `Scanner` class, which allows the user to interact with the `StrangeCalc` class. The reader is free to decide how the user will interact with the program.

Exercise 4.x)

Using the `HeadsOrTails` class defined in the solution of the Exercise 4.l, create a class called `HeadsOrTailsGame`, which simulates the tossing of a coin, and which allows the user to guess whether "heads" or "tails" will come out. The program will have to print a final message stating if the user has won or not.

Exercise 4.y)

Change the `HeadsOrTailsGame` class created in the previous exercise, so that the program initially allows you to specify the number of attempts to do. The program will have to count the number of times the user has guessed the result of the coin toss, and the number of times he has not guessed, and will have to decide whether he has won the game or not.

Exercise 4.z)

Considering the `PhoneBook` class created in Exercise 3.x, create a method called `searchContactsbyName(String name)` which takes as input a string that can represent a name or part of it, and must return an array of `Contact` objects that contain this string in its name. Also create a test class called `SearchContacts` which allows the user to specify the string to be passed as a search criterion to the `searchContactsByName()` method and which prints the search results.

Chapter 4
Exercise Solutions

Operators and Execution Flow Management

```java
public class PairTest {
  public static void main(String args[]){
    int i = 0;
    while (true)
    {
      i++;
      if (i > 10)
        break;
      if ((i % 2) != 0)
        continue;
      System.out.println(i);
    }
  }
}
```

In the `main()` method we first declare a variable `i` that acts as an index and that we initialize to `0`. Then we declare a `while` infinite loop, whose condition is always `true`. Within the loop we immediately increase by one unit the value of the variable `i`. Then we check if the value of the aforesaid variable is greater than `10`, if the answer is yes, the following `break` construct will

make us exit the loop that had to be infinite, and consequently the program will terminate immediately after. If the value is less than 10, using the modulo operator (%), checks if the rest of the division between i and 2 is different from 0. But this rest will be different from 0 if and only if i is an odd number. If therefore the number is odd, with the continue construct the flow will pass to the next iteration starting from the first statement of the while loop (i++). If i is an even number, then it will be printed.

The output of the previous program is the following:

```
2
4
6
8
10
```

Solution 4.b)

```java
public class ArrayTest {
    public static void main(String args[]) {
        for (int i = 0; i < 26; ++i) {
            char c = (char)('a' + i);
            System.out.println(c);
        }
    }
}
```

We execute a loop with the index i that varies from 0 to 25. Adding to the character 'a' the value of the index i (which at each iteration increases by one unit), we will get the other characters of the alphabet. The cast to char is necessary because the sum between a character and an integer is promoted to integer.

Solution 4.c)

The code could be the following:

```java
public class MultiplicationTables {
    public static void main(String args[]) {
        for (int i = 1; i <= 10; ++i) {
            for (int j = 1; j <= 10; ++j) {
                System.out.print(i*j + "\t");
            }
            System.out.println();
        }
    }
}
```

With this double nested loop and using the escape character \t, we can print the multiplication table with a few lines of code.

Solution 4.d) Operators and Execution Flow, True or False:

1. **True**.

2. **True**.

3. **False**, the boolean variable `flag` will be `true`. The *atomic* expressions have value true-false-true, respectively. Indeed, it's true that: i = 6, j = 7, k = 5, h = 6. In fact, (i != j) is true and also (i <= h) is true. The expression ((j <= k) || (i <= h))) is true. Finally, the AND operator cause the `flag` value is `true`.

4. **True**.

5. **False**, `switch` can only check an integer variable (or a compatible type) by comparing its equality with the constants. From version 5 we can also use enumerations and the `Integer` type (or a compatible type), and from version 7 also strings. The `if` construct allows to perform cross-checks using objects, boolean expressions, etc.

6. **False**, the ternary operator is equivalent to an expression that returns a value. In particular it always produces a value, and this must necessarily be assigned or used in some way (assigning it to a variable, passing it as an argument to a method, etc.). For example, if i and j are two integers, the following expression: i < j ? i : j; would cause a compilation error (besides not making sense), since the result is not used.

7. **True**.

8. **False**, the do loop, unlike the `while` loop, guarantees the execution of the first iteration on the code block in any case.

9. **False**, the `switch` is a condition construct, not a loop construct.

10. **False**, the `continue` cannot be used within a `switch` construct, but only within loops.

Solution 4.e)

The code could be the following:

```
public class ArgsTest {
    public static void main(String args[]) {
        if (args.length == 1) {
            System.out.println(args[0]);
```

```
        } else {
            System.out.println("Please, specify a value from the command line");
        }
    }
}
```

Solution 4.f)

This is because we do not know a priori how our program will evolve, and therefore, even if the default clause might not be needed when writing the program, modifying the program could create a new unexpected condition, creating a bug in our application. In fact, a new case will not be expected and the execution flow will not enter into any case clause of the switch construct. Even using the default construct just to print a sentence "Unexpected case" could be a good habit.

Solution 4.g)

The answer is identical to the previous one. The else clause for an if construct is equivalent to a default clause for the switch construct.

Solution 4.h)

The requested code could be the following:

```
public class Exercise4H {
    public static void main(String args[]) {
        for (int i = 1, j = 1; j <= 10; i++) {
            if (i % 3 == 0){
                System.out.println("Number multiple of 3 = " + i);
                j++;
            }
        }
    }
}
```

Solution 4.i)

The requested code could be the following:

```
public class Exercise4I {
    public static void main(String args[]) {
        int i = 1, j = 1;
        while(j <= 10)  {
            if (i % 3 == 0){
```

```
                System.out.println("Number multiple of 3 = " + i);
                j++;
            }
            i++;
        }
    }
}
```

Solution 4.j)

The requested code could be the following:

```java
public class Exercise4J {
    public static void main(String args[]) {
        int i = 1, j = 1;
        do {
            if (i % 3 == 0){
                System.out.println("Number multiple of 3 = " + i);
                j++;
            }
            i++;
        } while(j <= 10) ;
    }
}
```

Solution 4.k)

The required EvenOrOdd class code could be the following:

```java
import java.util.*;

public class EvenOrOdd {
    public String getEvenOrOdd() {
        Random random = new Random();
        return random.nextInt() % 2 == 0 ? "Even" : "Odd";
    }
}
```

> **Note that we used a ternary operator because it seems the best choice, but we could have used a simple `if` construct as well.**

While the EvenOrOddTest class code could be:

```java
public class EvenOrOddTest {
    public static void main(String args[]) {
        EvenOrOdd evenOrOdd = new EvenOrOdd();
        System.out.println(evenOrOdd.getEvenOrOdd());
    }
}
```

Solution 4.l)

The code of the requested HeadsOrTails class could be the following:

```java
public class HeadsOrTails {
    public String getHeadsOrTails() {
        EvenOrOdd evenOrOdd = new EvenOrOdd();
        String evenOrOddString = evenOrOdd.getEvenOrOdd();
        String headsOrTails = switch (evenOrOddString) {
            case "Even" -> "Heads";
            case "Odd" -> "Tails";
            default -> "There's a Bug!!!";
        };
        return headsOrTails;
    }
}
```

while the class code that HeadsOrTailsTest could be the following:

```java
public class HeadsOrTailsTest {
    public static void main(String args[]) {
        HeadsOrTails headsOrTails = new HeadsOrTails();
        System.out.println(headsOrTails.getHeadsOrTails());
    }
}
```

Solution 4.m)

The requested code could be the following:

```java
import java.util.Scanner;

public class Moderator {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String string = "";
        System.out.println("Type something then press enter, or type"
                            + " \"end\" to end the program");
        while (!(string = scanner.next()).equals("end")) {
            string = moderateString(string);
```

```
              System.out.println("You typed " + string.toUpperCase() + "!");
        }
        System.out.println("Program ended!");
    }

    private static String moderateString(String string) {
        switch (string) {
            case "gosh":
            case "golly":
            case "hilarious":
            case "jocund":
                string = "CENSORED!";
                break;
            default:
                break;
        }
        return string;
    }
}
```

It's just one of the solutions (certainly not the most elegant).

Solution 4.n)

Let's list all the cases:

1. + no, if used as a sum operator, but as a string concatenation operator it allows to concatenate a string to a boolean.

2. % no.

3. ++ no.

4. /= no.

5. & yes, i.e. (true & false) = true.

6. = yes, is an assignment operator, applicable to any type.

7. !! this is not a valid operator!

8. >>> no.

Solution 4.o)

The correct answer is: 99.
In fact, i is initially 99. Then in the boolean condition of the first if, a post-increment operator

95

is used, which having lower priority than the >= operator is executed after it. This implies that i still holds 99 when tested if it is >=100, and only after this check is incremented to 100. So, the if condition is false, and the related code block is not executed. Then the code block of the else clause is executed. Here the result of a ternary operator is printed. In fact, the value of i is decreased from 100 to 99, and therefore the ternary operator returns the first value, that is i++. Also in this case, it is a post-increment operator (with low priority), and therefore the value of i (99) is first printed and then the variable is incremented (but the program terminates immediately after).

Solution 4.p)

The correct answer is: 32 and at the next line 2, or the output is the following:

```
32
2
```

In fact, initially i is 22, and j is as the remainder of 22 (and not 23 because the post-increment is applied after the modulo operator %) divided by 3, or 1. After that, is assigned to i the return value of the ternary operator which checks if j!=0 (that is true). Then the value of j is returned which is 1. The execution flow, in the switch construct enters in the case 1 where 8 << 2 is printed, which is equivalent to 8 multiplied by 2 raised to 2, or 32. The case 2 is also executed, since there is no break that at the end of the case 1. Then 8 >> 2 is printed which is equivalent to 8 divided by 2 raised to 2, or 2.

Solution 4.q)

The code could be the following:

```java
import java.util.Scanner;

public class Exercise4Q {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println(
            "Type the number of days passed from the end of your last holidays");
        int days = scanner.nextInt();
        System.out.println("You typed " + days + " days!");
        int ore = days*24;
        int minutes = ore*60;
        System.out.println("So " + minutes +
            " minutes are just passed from your last holidays!");
    }
}
```

Solution 4.r)

The solution could be the following:

```java
public class Exercise4R {

    private static int matrix[][] = {
        {1, 7, 3, 9, 5, 3},
        {6, 2, 3},
        {7, 5, 1, 4, 0},
        {1, 0, 2, 9, 6, 3, 7, 8, 4}
    };

    public static void main(String args[]) {

        int numberToFind = Integer.parseInt(args[0]);

        FIRST_LABEL:
        for (int i = 0; i < matrix.length; i++) {
            int[] row = matrix[i];
            for (int j = 0; j < row.length; j++) {
                if (row[j] == numberToFind) {
                    System.out.println(numberToFind + " found at row = "
                        + ++i + ", column = " + ++j);
                    break FIRST_LABEL;
                }
            }
        }

        System.out.println("Search completed");
    }
}
```

We first had to convert `args[0]` using the static method of the `Integer` class `parseInt()` (see Exercise 3.t) storing it in a variable `numberToFind`. Then we used a double nested loop to navigate inside the cells of the matrix, using the indices `i` (for the rows) and `j` (for the columns). Note the use of the label we called `FIRST_LABEL`, which marks the external loop. When the first occurrence of the `numberToFind` is found, the instruction:

```java
break FIRST_LABEL;
```

terminates the external loop, and the program continues printing the message `Search completed`, and then ends.

Solution 4.s)

The code could be the following:

```java
public class Exercise4S {

    private static int matrix[][] = {
        {1, 7, 3, 9, 5, 3},
        {6, 2, 3},
        {7, 5, 1, 4, 0},
        {1, 0, 2, 9, 6, 3, 7, 8, 4}
    };

    public static void main(String args[]) {

        int numberToFind = checkArgument(args);

        if (numberToFind == -1) {
            System.out.println("Specificy an integer number between 0 and 9");
            return;
        }

        FIRST_LABEL:
        for (int i = 0; i < matrix.length; i++) {
            int[] row = matrix[i];
            for (int j = 0; j < row.length; j++) {
                if (row[j] == numberToFind) {
                    System.out.println(numberToFind + " found at row = "
                            + ++i + ", column = " + ++j);
                    break FIRST_LABEL;
                }
            }
        }

        System.out.println("Search completed");
    }

    private static int checkArgument(String[] args) {
        if (args.length == 1) {
            if (args[0].length() == 1) {
                for (int i = 0; i < 10; i++) {
                    if (args[0].equals("" + i)) {
                        return Integer.parseInt(args[0]);
                    }
                }
            }
        }
        return -1;
    }
}
```

Note that we have delegated to the checkArgument() method the correctness of the input specified by the user. This method returns the specified value, or, if it is not correct, the value -1. As you can see from the code of the main() method, if the value returned by the checkArgument() method is -1, a help message is printed for the user, and with the return statement, the method ends. Note that the main() method returns void, so to exit the method we use the return statement without specifying what to return.

Let's now analyze the method checkArgument(). With the first if construct, we first checked that the length of the array args is 1, or that a single argument has been specified, using the length variable of the array (see section 3.6.5). With the second if construct, we checked that the length of the string args[0] is exactly 1. We used the call to the length() method of the String class (not to be confused with the variable length of the array). The following for loop executes a loop on values ranging from 0 to 9, and checks that args[0] coincides with one of the values. When it finds a match, the current value is returned after having converted it to an integer by calling the static method of the Integer class parseInt() (see Exercise 3.t). If, on the other hand, there is no correspondence in the for loop, for example because a letter or symbol has been specified (so not an integer between 0 and 9), then the loop will end and the value -1 will be returned.

Solution 4.t)

The SimpleCalc requested class could be similar to the following:

```java
public class SimpleCalc {

    public double sum(double d1, double d2) {
        return d1 + d2;
    }

    public double subtract(double d1, double d2) {
        return d1 - d2;
    }

    public double multiply(double d1, double d2) {
        return d1 * d2;
    }

    public double divide(double d1, double d2) {
        return d1 / d2;
    }

    public double returnRest(double d1, double d2) {
        return d1 % d2;
    }
```

```java
    public double maximum(double d1, double d2) {
        return d1 > d2 ? d1 : d2;
    }

    public double minimum(double d1, double d2) {
        return d1 > d2 ? d2 : d1;
    }

}
```

While the test class could be:

```java
public class Exercise4T {

    public static void main(String args[]) {
        SimpleCalc simpleCalc = new SimpleCalc();
        System.out.println("42.7 + 47.8 = " +
            simpleCalc.sum(42.7, 47.8));
        System.out.println("42.7 - 47.8 = " +
            simpleCalc.subtract(42.7, 47.8));
        System.out.println("42.7 x 47.8 = " +
            simpleCalc.multiply(42.7, 47.8));
        System.out.println("42.7 : 47.8 = " +
            simpleCalc.divide(42.7, 47.8));
        System.out.println("the rest of the division between 42.7 and 47.8 è " +
            simpleCalc.returnRest(42.7, 47.8));
        System.out.println("the maximum between 42.7 and 47.8 è " +
            simpleCalc.maximum(42.7, 47.8));
        System.out.println("the minimum between 42.7 and 47.8 è " +
            simpleCalc.minimum(42.7, 47.8));
    }
}
```

Solution 4.u)

The requested code could be the following:

```java
import java.util.*;

public class Exercise4U {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Type the first operand then press enter.");
        double firstOperand = Double.parseDouble(scanner.nextLine());
        System.out.println(
            "Now choose the operation to perform then press enter:");
        printOperationsTable();
```

```
        String operation = scanner.nextLine();
        System.out.println("Now choose the second operand then press enter.");
        double secondOperand = Double.parseDouble(scanner.nextLine());
        double result=performOperation(firstOperand, secondOperand, operation);
        System.out.println("Result = " + result);
    }

    private static double performOperation(double firstOperand,
                                    double secondOperand, String operation) {
        SimpleCalc simpleCalc = new SimpleCalc();
        switch (operation) {
            case "+":
                return simpleCalc.sum(firstOperand, secondOperand);
            case "-":
                return simpleCalc.subtract(firstOperand, secondOperand);
            case "x":
                return simpleCalc.multiply(firstOperand, secondOperand);
            case "d":
                return simpleCalc.divide(firstOperand, secondOperand);
            case "r":
                return simpleCalc.returnRest(firstOperand, secondOperand);
            case "u":
                return simpleCalc.maximum(firstOperand, secondOperand);
            case "m":
                return simpleCalc.minimum(firstOperand, secondOperand);
            default:
                System.out.println("The operation specified " + operation +
                                    " is invalid");
                System.exit(1);
                return Double.NaN;
        }
    }

    private static void printOperationsTable() {
        System.out.println("'+' : sum");
        System.out.println("'-' : subtract");
        System.out.println("'x' : multiply");
        System.out.println("'d' : divide");
        System.out.println("'r' : return the rest of the division");
        System.out.println("'u' : maximum");
        System.out.println("'m' : minimum");
    }
}
```

The exercise had already outlined how to implement the interaction between the user and the program. In the next chapter we will see, among other things, how to find design solutions to our programs, and how important it is to do some activities before starting to code. For the rest, the code is quite clear, and we will make just some observations on the most obscure points.

Note that to perform the calculations we used the widest numeric data type: the double type. Note also the use of the static method parseDouble() of the Double class, which converts a string to double (see official documentation), as well as the parseInt() method of the Integer class converts a string to an int (see Exercise 3. t).

We have not managed (it was not required) the possible incorrect use of the program by the user, because it is too demanding for the notions we have studied so far. With the exception handling that we will study in Chapter 9, we will find simple solutions. Unfortunately, in some operations it loses its precision. You can check it for example by performing an operation of remainder from the division between 2.3 and 2, which results in an inaccurate value as you can see from this output example:

```
Type the first operand then press enter.
2.3
Now choose the operation to perform then press enter:
'+' : sum
'-' : subtract
'x' : multiply
'd' : divide
'r' : return the rest of the division
'u' : maximum
'm' : minimum
r
Now choose the second operand then press enter.
2
Result = 0.299999999999998
```

This is due to how the double type in memory is represented, and we had also mentioned it in section 3.3.2.1 (we should use the BigDecimal type, see official documentation), and it is a problem common to all modern programming languages that use the same method of representation in memory (IEEE-754 Standard). The System.exit() method terminates the program instantly. It should also be noted that the program proposes to the user to choose specific letters (see performOperation() method) that correspond to operations to be performed.

Solution 4.v)

The StrangeCalc requested class could be similar to the following:

```java
public class StrangeCalc {

    public double sum(double... doubles) {
        double result = 0;
        for (double aDouble : doubles) {
            result += aDouble;
        }
        return result;
    }
}
```

```
    public double multiply(double... doubles) {
        double result = doubles[0];
        for (int i = 1; i < doubles.length; i++) {
            result *= doubles[i];
        }
        return result;
    }

    public double maximum(double... doubles) {
        double max = doubles[0];
        for (int i = 1; i < doubles.length; i++) {
            double aDouble = doubles[i];
            if (aDouble > max) {
                max = aDouble;
            }
        }
        return max;
    }

    public double minimum(double... doubles) {
        double min = doubles[0];
        for (int i = 1; i < doubles.length; i++) {
            double aDouble = doubles[i];
            if (aDouble < min) {
                min = aDouble;
            }
        }
        return min;
    }
}
```

We used varargs as parameters to make the calling of these methods easier. The sum() method is very simple and a simple enhanced for loop is used to execute the sum. In the other three cases, we had to first recover the first element, and then carry out the operations with the rest of the elements of the array passed as input.
While the test class could be the following:

```
public class Exercise4V {

    public static void main(String args[]) {
        StrangeCalc strangeCalc = new StrangeCalc();
        System.out.println("42.7 + 47.8 = " + strangeCalc.sum(42.7, 47.8));
        System.out.println("42.7 x 47.8 x 2= " +
            strangeCalc.multiply(42.7, 47.8, 2));
        System.out.println("The maximum between 42.7, 47.8, 50, 2, 8, 89 is " +
            strangeCalc.maximum(42.7, 47.8, 50, 2, 8, 89));
        System.out.println("The minimum between 42.7, 47.8, 50, 2, 8, 89 is " +
            strangeCalc.minimum(42.7, 47.8, 50, 2, 8, 89));
    }
}
```

Solution 4.w)

The requested code could be the following:

```java
import java.util.*;

public class Exercise4Z {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Type an operand, then press enter " +
            "(add other operands repeating this operation). \nWhen you are " +
            "finished choose the operation to be performed, then press enter.");
        printOperationsTable();
        String temp;
        String operandsString = "";
        while (isNotOperation(temp = scanner.nextLine())) {
            operandsString += temp + "-";
        }
        if (isNotOperation(temp)) {
            System.out.println("Operation code error!");
        }
        String[] operandsArray = operandsString.split("-");
        double[] operands = new double[operandsArray.length];
        for (int i = 0; i < operandsArray.length; i++) {
            operands[i] = Double.parseDouble(operandsArray[i]);
        }
        double result = performOperation(operands, temp);
        System.out.println("Result = " + result);
    }

    private static boolean isNotOperation(String line) {
        if (line.equals("+") || line.equals("x") || line.equals("u") ||
            line.equals("m")) {
            return false;
        }
        return true;
    }

    private static double performOperation(double[] operands, String operation) {
        StrangeCalc strangeCalc = new StrangeCalc();
        switch (operation) {
            case "+":
                return strangeCalc.sum(operands);
            case "x":
                return strangeCalc.multiply(operands);
            case "u":
                return strangeCalc.maximum(operands);
            case "m":
```

```
                    return strangeCalc.minimum(operands);
            default:
                System.out.println("Operation specified " + operation +
                                    " invalid!");
                System.exit(1);
                return Double.NaN;
        }
    }

    private static void printOperationsTable() {
        System.out.println("'+' : sum");
        System.out.println("'x' : multiply");
        System.out.println("'u' : maximum");
        System.out.println("'m' : minimum");
    }
}
```

We have decided to have the user specify all the operands, and then execute the operation when a possible operator is specified. Designing a solution to the problem was not trivial at all.

Also in this case, we have not managed (it was not required) the possible incorrect use of the program by the user, because it is too demanding for the notions that we have studied so far. With the exception handling that we will study in Chapter 9, we will find simple solutions.

The critical point of the code concerns the management of the input, which must be analyzed and transformed into data types that are used to perform our operations (double). Unfortunately, we are missing a very important topic that we have not yet studied: the **collections** (which we will examine in more detail in chapter 18, but we will also introduce them in chapters 8 and 12). Without this topic, to store the various operands specified by the user, we had use a string (operandsString) that contained the various operands separated by the dash symbol. Then with the split() method (see the String class documentation), we obtained an array of operands in the form of a string (operandsArray). Then we instantiated an array of double named operands of the same size as operandsArray, and filled it with double-type operands after converting them using the static parseDouble() method of the Double class.

A rather artificial solution, but it worked.

Here is an example of application execution:

```
Type an operand, then press enter (add other operands repeating this operation).
When you are finished choose the operation to be performed, then press enter.
'+' : sum
'x' : multiply
'u' : maximum
'm' : minimum
2
3
4
5
```

```
6
7.28
x
Result = 5241.6
```

Solution 4.x)

The required code could be the following:

```java
import java.util.*;

public class HeadsOrTailsGame {

    public static void main(String args[]) {

        System.out.println("Heads or Tails?");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();

        if ("heads".equals(input)) {
            System.out.println("Ok, tossing the coin...");
            HeadsOrTails headsOrTails = new HeadsOrTails();
            String result = headsOrTails.getHeadsOrTails();
            System.out.print("It's " +  result + "...");
            System.out.println("Heads".equalsIgnoreCase(result)? "you win!" :
                "you loose!");
        } else if ("tails".equals(input)) {
            System.out.println("Ok, tossing the coin...");
            HeadsOrTails headsOrTails = new HeadsOrTails();
            String result = headsOrTails.getHeadsOrTails();
            System.out.print("It's " +  result + "...");
            System.out.println("Tails".equalsIgnoreCase(result)? "you win!" :
                "you loose!");
        } else {
            System.out.println("I'm sorry, you can only write heads or tails,"
                + " try again...");
            System.out.println("Program terminated... bye!");
        }
    }
}
```

The algorithm implemented seems to work, unfortunately the same code is repeated several times. For now, it's okay because we've achieved our goal, but in the next chapters we'll try to improve the quality of our code.

Solution 4.y)

The required code could be the following:

```java
import java.util.*;

public class HeadsOrTailsGame {
    public static void main(String args[]) {
        System.out.println("Let's play heads or tails, how many attempts you"
            + " want to do?");
        Scanner scanner = new Scanner(System.in);
        String numberOfAttemptsString = scanner.nextLine();
        int numberOfAttempts = Integer.parseInt(numberOfAttemptsString);
        int counter = 1;
        var numberOfWinningAttempts = 0;
        var numberOfLosingAttempts = 0;
        String message = "";
        System.out.println("You have chosen to do "+ numberOfAttempts
            +" attempts...let's begin!");
        while (counter <= numberOfAttempts) {
            System.out.println("Attempt number " + counter);
            System.out.println("Heads or tails?");
            String choice = scanner.nextLine();
            if ("heads".equalsIgnoreCase(choice)) {
                System.out.println("Ok, tossing a coin...");
                HeadsOrTails headsOrTails = new HeadsOrTails();
                String result = headsOrTails.getHeadsOrTails();
                counter++;
                System.out.print("It's "+ result + "...");
                if ("heads".equalsIgnoreCase(result)) {
                        message = "you win!";
                        numberOfWinningAttempts++;
                } else {
                        message = "you lose!";
                        numberOfLosingAttempts++;
                }
            } else if ("tails".equalsIgnoreCase(choice)) {
                System.out.println("Ok, tossing a coin...");
                HeadsOrTails headsOrTails = new HeadsOrTails();
                String result = headsOrTails.getHeadsOrTails();
                counter++;
                System.out.print("It's "+ result + "...");
                 if ("tails".equalsIgnoreCase(result)) {
                        message = "you win!";
                        numberOfWinningAttempts++;
                } else {
                        message = "you lose!";
                        numberOfLosingAttempts++;
                }
```

```
        } else {
            message = "I'm sorry, you can only write heads or tails, try"
                + " again...";
        }
        System.out.println(message);
    }
    message = "You win " + numberOfWinningAttempts + " times, and lost "
        + numberOfLosingAttempts + " times, so...";
    if (numberOfWinningAttempts > numberOfLosingAttempts) {
        message += "you win the game! Congratulations!";
    } else if (numberOfWinningAttempts < numberOfLosingAttempts) {
        message += "you lose the game! Ah ah!";
    } else {
        message += "you draw the game! Try again!";
    }
    System.out.println(message);
    }
}
```

The code is quite understandable, and the reader should be able to interpret it. Note that we have used the equalsIgnoreCase() method of the String class to make comparisons without taking uppercase and lowercase letters into account. Furthermore, we can note the management of the counter variable, which is increased only when the launch is actually executed, but not when the launch is not executed if the user input is not compatible with the logic of the program.

Solution 4.z)

Let's insert the required method (in bold) into the PhoneBook class:

```
public class PhoneBook {
    public Contact [] contacts;
    public PhoneBook () {
        contacts = new Contact[]{
            new Contact("Claudio De Sio Cesari", "13, Java Street",
                "131313131313"),
            new Contact("Stevie Wonder", "10, Music Avenue", "1010101010"),
            new Contact("Gennaro Capuozzo", "1, Four Days of Naples Square",
                "1111111111")};
    }

    public Contact[] searchContactsByName(String name) {
        Contact []foundContacts = new Contact[contacts.length];
        for (int i = 0, j = 0;  i < foundContacts.length; i++) {
            if (contacts[i].name.toUpperCase().contains(name.toUpperCase())) {
                foundContacts[j] = contacts[i];
                j++;
```

```
            }
        }
        return foundContacts;
    }
}
```

The algorithm is not trivial, and also uses a for loop that uses two indexes (i and j), which are used for the two arrays involved. Note that the returned array (foundContacts) will be the same size as the contacts array, even if its elements may not be initialized. Also note that before using the contains() method of the String class to check if a contact name contains the name method parameter, we used the toUpperCase() method to make a non-case-sensitive comparison.

The SearchContacts class requested, could be:

```
import java.util.Scanner;

public class SearchContacts {
    public static void main(String args[]) {
        System.out.println("Search Contacts");
        System.out.println();
        var phoneBook = new PhoneBook();
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Contact[] foundContacts = phoneBook.searchContactsByName(input);
        System.out.println("Contacts found with name containing \""
            + input + "\"");
        for (Contact contact : foundContacts) {
            if (contact != null) {
                contact.printDetails();
            }
        }
    }
}
```

Note that the elements of the returned array are printed, only if different from null.

Chapter 5
Exercises

Real Development with Java

Here you will find many exercises a little different from the usual programming exercises. Learning to program with objects is a daunting task, but once you understand how to approach, you can get unimaginable results.

Exercise 5.a) JShell, True or False:

1. JShell is a program located in the **bin** directory of the JDK, like the **java** and **javac** tools, and we can call it directly from the command line, because we have correctly set the PATH variable to the **bin** folder to which it belongs.

2. JShell is an IDE.

3. A package cannot be declared in a JShell session.

4. The termination symbol ";" of a statement, can be omitted only if we write a Java statement on a single line.

5. Declaring the same variable twice is possible, because JShell follows different rules than the compiler. The last variable declared will overwrite the previous one.

6. If we declare a variable of type `String` without initializing it, it will be initialized automatically to `null`.

7. It is not possible to declare an interface in a JShell session.

8. The abstract modifier is always ignored within a JShell session.

9. It is not possible to declare a main() method in a JShell session.

10. You can declare annotations and enumerations in a JShell session.

Exercise 5.b) JShell Commands, True or False:

1. To end a JShell session, type the command goodbye.

2. All JShell commands must have the symbol \ as prefix.

3. The commands help and ? are equivalent.

4. The history command shows all the snippets and all the commands executed by the user in the current session. Next to the snippets there is a snippet id that allows you to recall the corresponding snippet.

5. The list command shows all the commands entered by the user in the current session.

6. The /types -all command will list all the variables declared in the current session.

7. The reload command will cause all instructions executed in the current session to be executed again.

8. The drop command can delete a certain snippet by specifying its snippet id.

9. If we specify the /set start JAVASE command, we will import all the Java Standard Edition packages into this and all other future JShell sessions.

10. With the command /! we recall the last edited snippet, be it valid or invalid.

Exercise 5.c)

Consider the following lines edited within a JShell session:

```
jshell> public int a;
a ==> 0

jshell> private String a
a ==> null

jshell> /reset
|   Resetting state.

jshell> /list
```

What will be the output of the final list command?

Exercise 5.d)

Considering all the instructions of the previous exercise, what will be the output of the following command?

```
jshell> /history
```

Exercise 5.e)

If we wanted to open in a JShell session the **HelloWorld.java** file (created in the first chapter) which is in the current directory, what command should we execute?

1. `/save HelloWorld.java`

2. `/retain HelloWorld.java`

3. `/reload HelloWorld.java`

4. `/open HelloWorld.java`

5. `/start HelloWorld.java`

6. `/env HelloWorld.java`

7. `/!`

Exercise 5.f)

Once the **HelloWorld.java** file is opened in a JShell session, how can we have the `Hello World!` string printed?

Exercise 5.g)

Which command we need to execute if we want to copy the **HelloWorld.java** file opened in the Exercise 5.e to the `C:/myFolder` folder?

1. `/save HelloWorld.java`

2. `/retain HelloWorld.java`

3. `/save HelloWorld2.java`

4. `/save C:/myFolder/HelloWorld.java`

5. `/save -start start C:/myFolder/HelloWorld.java`

6. `/env C:/myFolder/HelloWorld.java`

7. `/! C:/myFolder/HelloWorld.java`

Exercise 5.h) JShell Auxiliary Tools, True or False:

1. In a JShell session it is possible to declare a variable without specifying a reference.

2. In a JShell session it is possible to declare a reference without specifying the reference type.

3. In a JShell session it is possible to write a value, then with the **TAB** key to automatically infer the type of the variable to JShell, and then write the name of the reference.

4. While declaring a reference to a type that is not imported, it is possible to have JShell suggest a list of possible imports to use for the type, by simultaneously pressing the **SHIFT** and **TAB** keys, release them and then press the **v** key.

5. The **/edit** command will open the Notepad++ program

6. In a JShell session the simultaneous pressing of the **CTRL - SPACEBAR** keys, causes the auto-completion of the code you started writing, or in the case of more options available, the choice between them.

7. In a JShell session, pressing the **CTRL - E** keys at the same time, causes the cursor to move to the end of the line.

8. In a JShell session, pressing the **ALT - D** keys at the same time, causes the word to be deleted to the right of the cursor.

9. The `/set feedback silent` command, causes JShell to avoid printing the analysis messages on the written code.

10. Writing `System` and pressing the **TAB** key twice, JShell will show us the documentation of the `System` class.

Exercise 5.i)

The use of an IDE implies (choose all the answers that are believed to be correct):

1. Initial slowdown in learning development, because it requires the study of the same IDE.

2. The possibility of using a debugger.

3. The ability to integrate with other development tools, databases or application servers.

4. The possibility of using an advanced editor that allows us to automate the creation of programming constructs and use code refactoring techniques.

5. The inability to use packages, which must be managed from the command line.

Exercise 5.j)

Which of the following statements are correct?

1. From the version 10 of Java we can directly launch a source file containing a public class.

2. If we declare multiple classes within the same file, only one must be public in order to compile the file.

3. If we declare multiple classes within the same source file, only one must be public in order to launch the source file.

4. If we declare multiple classes within the same source file, only one must contain the `main()` method.

5. If we declare multiple classes within the same source file, the first must contain the `main()` method.

Exercise 5.k)

Which of the following statements are correct?

1. When we launch a source file, no **.class** file is generated.

2. Before a source file is executed, it is compiled by the JVM in memory.

3. You can launch a shebang file from the Windows command prompt.

4. It is possible to pass parameters to a shebang file

Exercise 5.l)

The architecture takes care of (choose all the answers you think are correct):

1. To define UML activity diagrams that model the functionalities of the system.

2. Improve software performance.

3. To optimize the use of resources by the software.

4. To optimize the partitioning of the software in such a way as to simplify the installation procedure.

Exercise 5.m)

A deployment diagram (choose all the statements that you think are correct):

1. It is a static diagram.

2. Show application execution flows.

3. Shows how hardware nodes host software components.

4. Highlight the dependencies between software components.

5. The main element of the diagram is a *node*, which is represented by a rectangle with two small rectangles emerging from the upper left corner.

Exercise 5.n)

We stated that the basic knowledge of topics such as XML and database, is essential for working in an IT company. This is because almost all of the applications use these two technologies in some way. If you work at a web application, what are the basic knowledge you need to have?

Exercise 5.o)

Briefly define the concepts of client, server, standalone application, mobile, web, web client, web server and enterprise application.

Exercise 5.p)

Define the tasks of the following corporate roles:

1. Project manager

2. Business analyst

3. IT manager

4. Release manager

116

5. DBA

6. Graphic designer

Exercise 5.q)

Defining what is an object-oriented methodology.

Exercise 5.r)

Which of the following statements are true about UML use case diagrams?

1. The class symbol cannot be used in the UML use case diagram.

2. A use case is represented with a rectangle in UML.

3. The use case diagram is a static diagram, in fact it does not contain information regarding the timeline.

4. The actors of the use case diagram represent the users of the system, and therefore natural persons.

5. A use case can be connected to multiple actors.

Exercise 5.s)

Which of the following statements are true?

1. A use case scenario does not contain conditions.

2. Scenarios can be described with an activity diagram as well as plain text.

3. All possible scenarios should be described for each use case.

4. The description of the scenarios allows us to discover any flows, before even writing the code.

5. A deployment diagram shows the interaction between the user and the system.

6. Candidate classes are the fundamental classes that will surely be implemented in our program.

Exercise 5.t)

Which of the following statements are true about UML sequence diagrams?

1. An activity diagram allows us to simulate a scenario, but using objects.

2. The activity line of an object is represented by a dotted line.

3. Messages represent methods.

4. In the activity diagram, it is possible to represent an actor.

5. The names of the objects in a sequence diagram are represented with the syntax `objectName:ClassName`, but one between the name of the object and the name of the class can be omitted.

Exercise 5.u)

Problem statement: create an authentication program that asks the user for username and password, and guarantees access to it (just printing a welcome message with the name of the user) if the credentials entered are correct. The system must support authentication for at least three username and password pairs.

Perform the use case analysis and identify the various use cases, following the advices of the sections 5.5.1. and 5.5.1.1.

> There are various software tools that support UML diagrams (https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools). Each of them has its philosophy and its complexity. You should choose one sooner or later (perhaps after having made some comparisons), although for now it is fine to use a sheet of paper, a pencil and an eraser. Choosing a tool, understanding how it works, etc. takes some time.

Exercise 5.v)

Identify the scenarios of the use cases of the previous exercise by following the advices of the section 5.5.1.2.

Exercise 5.w)

Let's continue the previous exercise following the process described in section 5.5.2. Being a simple desktop application, it seems superfluous to create a deployment diagram. But if we think about how we can one day reuse a part of this application (we are talking about an application that allows us to authenticate ourselves in a system, and that

118

we could also integrate in the case study introduced in Chapter 5 to authenticate in the Logos application), then it could be very useful to have another point of view, that specifies the dependencies between the various software components. So, let's try to create a component diagram (or a deployment diagram) by creating components that have significative names, and then we will begin to think to classes. It must be just a trivial diagram (high level deployment diagram) that enhances the possibility that a certain part of the software can be reusable.

Exercise 5.x)

Let's continue the previous exercise following the process described in section 5.5.3. So, we can identify the candidate classes, and consequently the key abstraction

Exercise 5.y)

Let's continue the previous exercise verifying the feasibility of the identified scenarios, creating sequence diagrams based on the interactions among the identified classes, as described in the section 5.5.4.

Exercise 5.z)

Based on the steps taken in the 5.r, 5.s, 5.t, 5.u, and 5.v exercises, implement a working solution.

Chapter 5
Exercise Solutions

Real Development with Java

1. **True**.

2. **False**.

3. **True**.

4. **True**.

5. **True**.

6. **True**, uninitialized variables are initialized to their own null values. A string, being an object, has its own `null` value.

7. **False**.

8. **False**.

9. **False**, it is possible to declare a `main()` method, but it will not have the same role of *initial method* as in an ordinary Java program.

10. **True**.

Solution 5.b)

1. **False**, we need to edit the `exit` command.

2. **False**, all JShell commands must be prefixed with `/`.

3. **True**.

4. **False**, it is true that the `history` command shows all the snippets and all the commands executed by the user in the current session, but it is not true that next to the snippets there is a snippet id.

5. **False**, the `list` command shows all the snippets entered by the user in this session, with the respective snippet id next to them.

6. **False**, the `/types -all` command it will list all the types (classes, interfaces, enumerations, annotations) declared in the current session. Rather, the `/ variables -all` command will list all the variables declared in the current session.

7. **True**.

8. **True**.

9. **False**, it is true that we will import all the Java Standard Edition packages in this session, but not in future sessions (we should also have explicitly specified the `-retain` option).

10. **True**.

Solution 5.c)

The output of the `list` command will be empty. In fact, the `reset` command will have reset all the entered snippets.

Solution 5.d)

The output of the `history` command will be the following:

```
public int a;
private String a
/reset
/list
/history
```

Solution 5.e)

The correct command is:

4. `/open HelloWorld.java`

Solution 5.f)

We can only invoke the `main()` method in the following way:

```
jshell> HelloWorld hw = new HelloWorld();
hw ==> HelloWorld@52a86356

jshell> hw.main(null);
Hello World!
```

Note that since the `args` array is not used within the `main()` method, then we could pass it `null`.
Even if we haven't studied it seriously yet, the `static` modifier allows us to avoid instantiating the object `hw`, and to execute the command directly using the class name:

```
jshell> HelloWorld.main(null)
Hello World!
```

Solution 5.g)

The correct command is:

4. `/save C:/myFolder/HelloWorld.java`

Solution 5.h)

1. **True**, that is an *implicit variable*, and JShell will automatically infer the type.

2. **True**, in this case we talk about *forwarding reference*, and JShell will create the reference, but it won't make it available until we declare its type as well. At that point the reference will be replaced and initialized to `null`.

3. **False**, instead, press the **SHIFT** and **TAB** keys simultaneously, release them and then press the **v** key (which stands for "variable"). JShell will infer the type of the variable, declare it and position the cursor immediately after it to allow us to define the reference.

4. **False**, instead, press the **SHIFT** and **TAB** keys simultaneously, release them and then press the **i** key (which stands for "input").

5. **False**, the JShell Edit Pad program will be opened, unless first we have set as default editor Notepad++ using the command:

```
/set editor C:\Program Files (x86)\Notepad++\notepad++.exe
```

6. **False**, pressing the **TAB** key causes the auto-completion of the code you started writing, or in the case of more options available, the choice between them.

7. **True**.

8. **True**.

9. **True**.

10. **True**.

Solution 5.i)

Only the fifth answer is false. An IDE, on the other hand, exempts the programmer from complex package management.

Solution 5.j)

Only the second and fifth answers are true. In particular, the first one is false because it is possible to directly launch a source file from version 11, not from version 10.

Solution 5.k)

Only the third answer is false.

Solution 5.l)

Only the first answer is false. An architect in fact dedicates himself essentially to the non-functional requirements of the application.

Solution 5.m)

The following statements are incorrect:

1. number 2: because no execution flow is shown.

2. number 5: it is true that the *node* is the main element of the diagram, but is represented as a three-dimensional cube. The description instead refers to the *component* element.

Solution 5.n)

If you work in the web area, you have basic knowledge about the HTTP protocol, HTML, Javascript and CSS languages, libraries like Bootstrap, and frameworks like Spring, or Angular JS and so on.

Solution 5.o)

By definition, a *client* is a program that requires services to another program called a server. By definition, a *server* is a program that is always running, which provides services. A client and a server usually communicate over the network, with a well-defined protocol.

Standalone applications (also called *desktop applications*) are run on desktops and laptops, and usually have a graphical interface.

Web applications are applications that have an architecture divided into a client part and a server part.

A *web client* requests web pages, and coincides with the programs that we commonly call browsers (Mozilla Firefox, Google Chrome and so on).

A *web server* is instead a server application that provides services available on the net.

Enterprise applications are an evolution of web applications, and usually provide more complex services such as downloading resources, web services (i.e. communication applications between heterogeneous systems using the HTTP protocol), services reporting, etc., and therefore as an *enterprise client* can also have programs specially created to interact with the *enterprise server* layer. The latter is in turn made up of various layers that use different technologies to fulfil various purposes.

A *mobile application* (often simply called an *app*) is an application that runs on mobile clients, such as smartphones and tablets, and can even have a server counterpart.

Solution 5.p)

See section 5.4.4.

Solution 5.q)

An *object-oriented methodology*, in its most general definition, could be understood as a couple consisting of a process and a modelling language.

In turn, a *process* could be defined as a series of indications regarding the steps to be taken to successfully complete a project.

125

A *modelling language* is instead the tool that the methodologies use to describe (possibly in a graphic manner) all the static and dynamic characteristics of a project. The modelling language considered de facto standard is the UML.

Solution 5.r)

The only false answers are the number 2 and the number 4. In fact, as regards the statement number 2, the only rectangle that is part of the syntax of the use case diagram, is the system boundary, which usually surrounds the use cases and delimits the system. As for statement number 4, in reality the concept of actor must be interpreted as "role". A system user could also be another system.

Solution 5.s)

Only the last two statements are false. In particular, the definition reported in statement number 5 is about the use case diagram, not the deployment diagram. Instead, the definition reported in statement number 6 is that of key abstractions, not candidate classes.

Solution 5.t)

Only statement 2 is incorrect, as it reports the definition of an object's lifeline, and not an activity line.

Solution 5.u)

There is only one use case that we have called "Login" (see Figure 5.r.1). In fact, the interactions that the user will have with the system are limited to activities related to authentication, such as entering the username and password. Even with different types of flows that can be authenticated, the final task is solely to authenticate
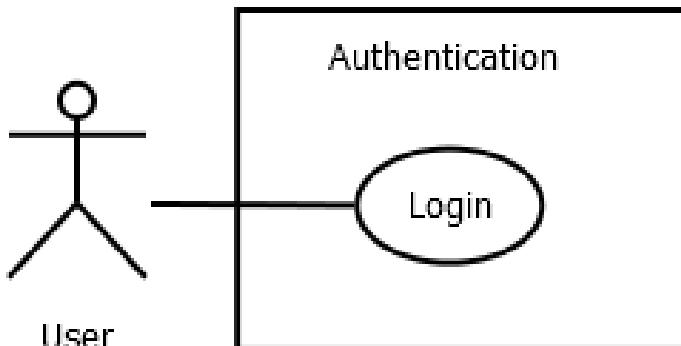
Figure 5.u.1 - Use case diagram.

126

> In our case we have found a single use case, but this does not mean that it will always be this way, and therefore that we must avoid the use case analysis! It is also essential for the most trivial programs

Solution 5.v)

The analysis of the scenarios is very subjective. The moment we write it, we are deciding "what" the application must do, something far from obvious.

Let's start from the prerequisite that we have not yet studied the graphical interfaces (to which the last two chapters are dedicated) and therefore we have to think about creating a program that works only from the command line.

Another prerequisite is that the system has statically preloaded some valid username and password pairs.

Main Scenario

1. The system asks the user to enter the username.

2. The user enters the username.

3. The system verifies that the username is valid and asks to enter the password.

4. The user enters the password.

5. The system checks that the password is valid and responds with a message confirming authentication, using the real name of the user who authenticated.

> As already said, this is only one of the possible solutions. We could also think of specifying together username and password, validating authentication with a captcha code, warning the user if the CAPS LOCK key is inserted, masking the password characters with asterisks, asking the user if he wants to memorize the username for the next login and so on. We have chosen a simple interaction.

Scenario 2

1. The system asks the user to enter the username.

2. The user enters the username.

3. The system does not recognize the entered username, prints a message and returns to step 1.

Scenario 3

1. The system asks the user to enter the username.

2. The user enters the username.

3. The system verifies that the username is valid, and asks to enter the password.

4. The user enters the password.

5. The system does not recognize the password entered, prints a message and returns to step 1.

6. By defining these three trivial scenarios it is much clearer what we have to do.

Solution 5.w)

With the diagram of Figure 5.t.1, we highlight how we will create a software component that contains the classes that perform the authentication, separated from the class that contains the `main()` method. The only tool we currently know to separate classes is packages, so we will use different packages.
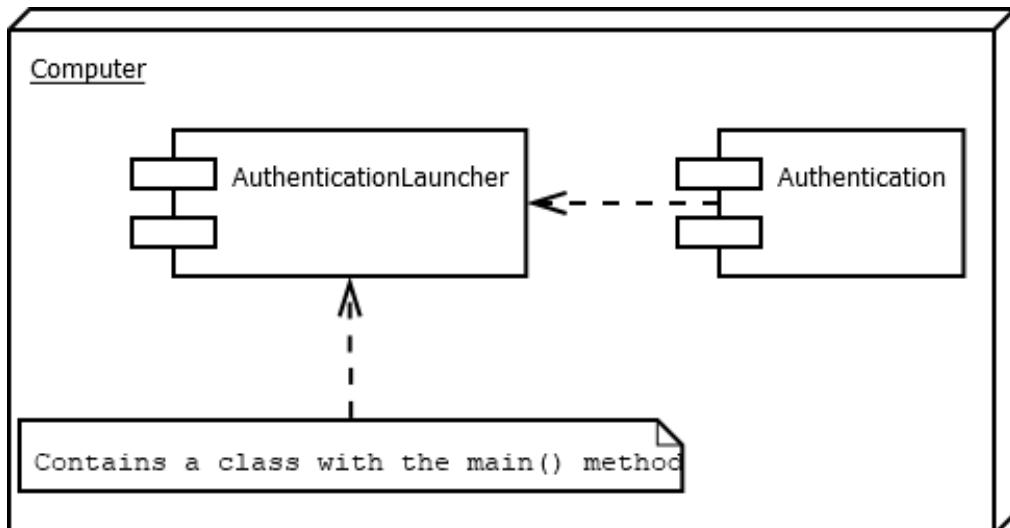
Figure 5.w.1 - Architecture of the application represented by a high level deployment diagram.

By consulting Appendix E, you can also learn how to create the JAR file containing the classes of the authentication component. In this way it will be easier to reuse it later as it is a single file. The Exercise E.a, relative to that appendix, requires the creation of the JAR file with the classes of this exercise.

Solution 5.x)

Following the process described in section 5.5.3, in order to find the list of key abstraction, we must first draw up the list of candidate classes. Below is our list with related comments.

1. **Authentication**: it could be a class to which give the responsibility to manage the main functionality of the application.

2. Login: it looks more like the name of the main application method. It could be a method within `Authentication` or an object declared in `Authentication`.

3. **User**: it could be the entity that contains the information for authentication.

4. System: too generic, does not seem to be the right name for a class.

5. Username: could be a property of the User class.

6. Password: it could be a property of the User class.

7. Name: could be a property of the User class.

8. Verification: it could be an `Authentication` method, or an object declared in `Authentication`.

9. Insertion: could define a method, but it does not seem a key abstraction.

10. Message: for now, it seems only a string, certainly not a key abstraction. This does not exclude that it could become a class later.

11. Substitution: it could define a method, but it does not seem a key abstraction.

So, the list of key abstractions, for now, is limited to the only two classes in the list in bold: `Authentication` and `User`. To these we add a `LauncherAuthentication` class which contains only the `main()` method and which has the sole responsibility of starting the application by instantiating the correct objects and recalling the methods appropriately.

> **Remember that the choice of these classes, like the previous steps, depend on experience, intentions, time available, predisposition, concreteness and the mindset of the person performing the analysis. There are thousands of solutions that can lead to development success, each of which has its pros and cons. The advice is to orientate (especially in the early days) on the simplest possible solution.**

Solution 5.y)

Following the process described in section 5.5.4, we now need to give the superficial definition of key abstraction, with interaction diagrams (see appendix G), i.e. collaboration or sequence diagrams.

These two diagrams are equivalent, so that many UML tools allow you to transform a collaboration diagram into a sequence diagram and vice versa pressing a button. In particular, a sequence diagram shows the interactions between the objects in a given period of time, emphasizing the sequence of messages that the entities exchange. A collaboration diagram, on the other hand, as the sequence diagram shows the interactions between objects in a given period of time, but emphasizes the structural organization of the interacting entities.

Since in this case it seems more interesting to emphasize the sequence of messages exchanged between objects, we will use a sequence diagram to describe the scenarios described in the solution of Exercise 5.s, using the objects described in Exercise 5.u. Since these latter objects are only key abstractions, at this time we can also decide whether we need to create new classes, add, modify or move methods, rename existing classes and so on. The diagram, with its "vision from the top", favors the identification of any incorrect situations, which can be improved or failed.

In Figure 5.y.1 we have only found a flow consistent with that described in the main scenario, using only the key abstraction identified in the previous exercise. The situation seen in this way seems to work.

The application user runs the application using the `LauncherAuthentication` class. This calls a method called `login()` on the `Authentication` class. From this point on, this method will perform operations. First it calls the `requestUsername()` method that requests the username from the user by printing a message, and waits for user input.

The application user runs the application using the `LauncherAuthentication` class. This calls a method called `login()` on the `Authentication` class. From this point on, this method will perform operations. First it calls the `requestUsername()` method that requests the username from the user by printing a message, and waits for user input.
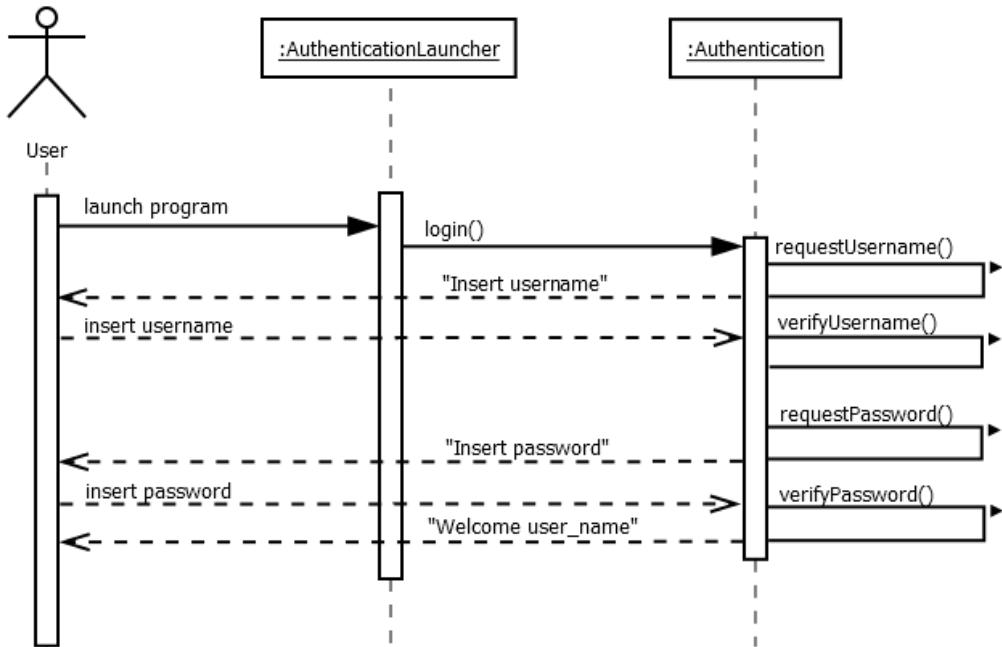
Figure 5.y.1 - Sequence diagram representing the main scenario.

> This method is internal, i.e. defined in the same class. This can be understood from the fact that the arrow that defines it, starts and ends in the same class. In UML the call of a method is indicated by of an "arrow" (which as a UML element is called a *message*) starting from a certain object. The "arrow" points to the object where the method called resides. So, an arrow that returns on the same object indicates the call to an internal method.

Then once the user enters the username, the Authentication class will call an internal method called verifyUsername(). This method will retrieve the user object from the user collection that has been defined to represent the list of users (which we will implement through arrays).

> This part was not represented in the diagram, to avoid make it too complex to read. We could have added a known element to specify our intentions, it would have been more correct, but we avoided since we have had to explain the diagram with these few lines.
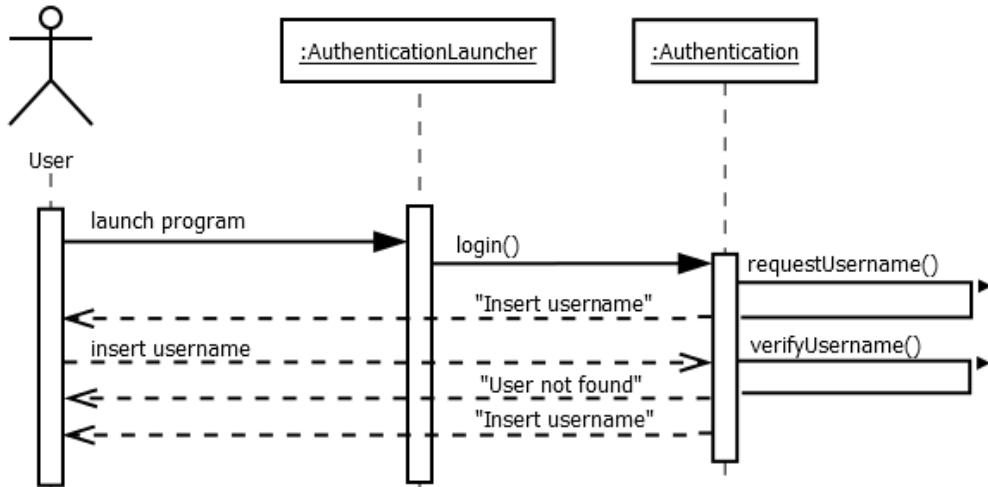
Figure 5.y.2 - Sequence diagram representing the second scenario.
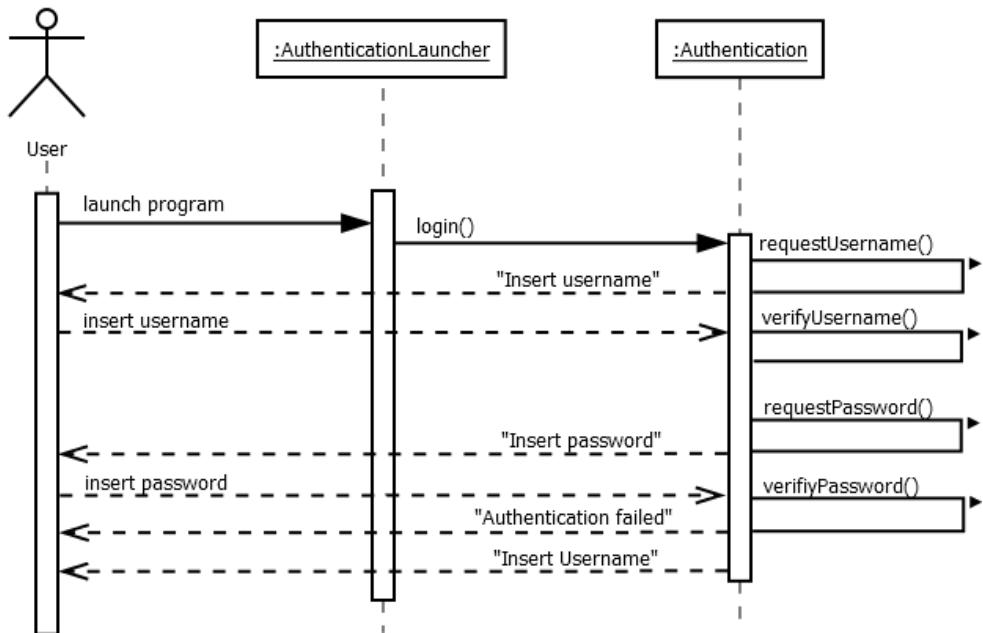
Figure 5.y.3 - Sequence diagram representing the third scenario.

So, even if the User object does not appear on the diagram, it is somehow involved. This is be-cause in our mind, a user should define the username, password and name variables, and then

to check if there is a certain username, but also to see if a password is associated with a certain username, a User object must be used.

The rest of the code is very simple to interpret. The internal method requestPassword() is called, which asks the user for the password by printing a message, and waits for user input. Then once the user enters the password, the Authentication class will call an internal method called verifyPassword() (and even in this case the verification will be done using a User object). Finally, the message "Welcome" is printed, specifying the name of the user who is authenticated (which is taken from the User object used to validate the authentication).

We can see that for how we designed the diagram in Figure 5.y.2, and in the related scenario, the username will be requested until a valid username is entered. The same goes for the password as it is possible to observe in Figure 5.y.3, which shows a sequence diagram related to the third scenario identified.

Solution 5.z)

The code that came out from our analysis is not exactly what we expected:

```java
package com.claudiodesio.authentication;

import java.util.*;

public class Authentication {

    private static final User[] users = {
        new User("Daniele", "dansap", "music"),
        new User("Giovanni", "giobat", "science"),
        new User("Ligeia", "ligder", "arte")
    };

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Insert username.");
            String username = scanner.nextLine();
            User user = verififyUsername(username);
            if (user == null) {
                System.out.println("User not found!");
                continue;
            }
            System.out.println("Insert password");
            String password = scanner.nextLine();
            if (password != null && password.equals(user.password)) {
                System.out.println("Welcome " + user.name);
                break;
            } else {
```

```
                        System.out.println("Authentication failed");
                }
            }
        }

    private static User verifyUsername(String username) {
        if (username != null) {
            for (User user : users) {
                if (username.equals(user.username)) {
                    return user;
                }
            }
        }
        return null;
    }
}
```

When we wrote the code, we found some difficulties. The first was to implement the algorithm described in the scenarios: we had to introduce an infinite loop and the break and continue commands, not really the solution we expected.

We have also taken another decision in contrast to our analysis: delete the LauncherAuthentication class which should have contained only the main() method. This was because it seemed a forced decision and lacked any use.

Another doubt arose when we had to verify the correctness of the username and password, as we had not decided in the design phase if the verification had to take into account capital or small letters.

We could not even create a verifyPassword() method complementary to the verifyUsername(), since if we had created a separate method we could not have used the break clause to exit the infinite loop. In short, compared to the analysis we did, there were problems that we solved directly with the code. But where did these problems come from? Why didn't our process work the way we guessed?

The answer is that essentially, we lack fundamental concepts that we have yet to study. Therefore, this exercise will continue in the exercises of the next chapter to make it clearer and more efficient. In particular, we have missed two basic steps: assigning responsibilities to the classes we create, and creating a class diagram that helps us better distribute responsibilities between classes. Each object must have a single responsibility, or several responsibilities closely related to each other. Responsibilities will be implemented either as methods or as variables, and define roles that can be assigned to objects.

In any case the program we wrote works correctly. Although our analysis was not perfect, he gave us some important indications. For example, the analysis of the scenarios was fundamental to understand what we had to do. And the interaction diagrams have also directed us

towards the implementation solution (then partially disregarded).

> **After the experience with these exercises, if you were not able to solve the exercises 4.m, 4.q, 4.r, 4.s, 4.t, 4.u, 4.ve 4.z of the previous chapter, you could try to redesign them from scratch and try to find a solution by yourself, maybe different from the one proposed.**

Chapter 6
Exercises

Encapsulation and Scope

It's time to slowly discover the right object-oriented mentality.

> **After each exercise, we recommend at least to consult the solution.**

Exercise 6.a) Object Orientation Theory, True or False:

1. The Object Orientation has been created only a few years ago.

2. Java is a non-pure object-oriented language, SmallTalk is a pure object-oriented language.

3. All object-oriented languages support object-oriented paradigms in the same way. It can be said that a language is object-oriented if it supports encapsulation, inheritance and polymorphism; in fact, other paradigms such as abstraction and reuse also belong to functional philosophy.

4. Applying abstraction means focusing only on the important characteristics of the entity to be abstracted.

5. The reality that surrounds us is a source of inspiration for the object-oriented philosophy.

6. Encapsulation helps us interact with objects; abstraction helps us interact with classes.

7. Reuse is favored by the implementation of other object-oriented paradigms.

8. Inheritance allows the programmer to manage multiple classes collectively.

9. Encapsulation divides objects into two separate parts: the public interface and the internal implementation.

10. To use the object, it is enough to know the internal implementation, it is not necessary to know the public interface.

Exercise 6.b) Encapsulate and complete the following classes:

```java
public class Driver {
    private String name;

    public Driver(String name) {
        // set the name
    }
}

public class Car {
    private String stable;
    private Driver driver;

    public Car(String stable, Driver driver) {
      // set the stable and the driver
    }

    public String getDetails() {
      // return a descriptive string of the object
    }
}
```

Keep in mind that the Car and Driver classes must then be used by the following classes:

```java
public class RaceTest {
    public static void main(String args[]) {
        Race monteCarlo = new Race("Montecarlo GP");
        monteCarlo.runRace();
        String result = monteCarlo.getResult();
        System.out.println(result);
    }
}

public class Race {
    private String name;
```

```
    private String result;
    private Car grid [];

    public Race(String name) {
        setName(name);
        setResult("Race not finished");
        createStartingGrid();
    }

    public void createStartingGrid() {
        Driver one = new Driver("Joey");
        Driver two = new Driver("Dee Dee");
        Driver three = new Driver("Johnny");
        Driver four = new Driver("Tommy");
        Car carNumberOne = new Car("Ferrari", one);
        Car carNumberTwo = new Car("Renault", two);
        Car carNumberThree = new Car("BMW", three);
        Car carNumberFour = new Car("Mercedes",  four);
        grid = new Car[4];
        grid[0] = carNumberOne;
        grid[1] =  carNumberTwo;
        grid[2] = carNumberThree;
        grid[3] =  carNumberFour;
    }

    public void runRace() {
        int winnerNumber = (int)(Math.random()*4);
        Car winner = grid[winnerNumber];
        String result = winner.getDetails();
        setResult(result);
    }

    public void setResult(String winner) {
        this.result = "Winner of " + this.getName() + ": " + winner;
    }

    public String getResult() {
        return result;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Exercise analysis

The `RaceTest` class contains the `main()` method and therefore defines the application execution flow. It is very readable: we can instantiate a `race` object and call it "Montecarlo GP", run the race, request the result and print it.

The `Race` class, on the other hand, contains few simple methods and three instance variables: `name` (the name of the race), `result` (a string containing the name of the winner of the race if it was run) and `grid` (an array of `Car` objects that participate to the race).

The constructor takes as input a string with the name of the race that is appropriately set. Furthermore, the value of the `result` string is set to "Run not completed". Finally the method `createStartingGrid()` is called.

The `createStartingGrid()` method instantiates four `Driver` objects by assigning them names. It then instantiates four `Car` objects by assigning them the names of the stables and their drivers. Finally instantiate and initialize the grid array with the newly created cars. A race, after being instantiated, is ready to run.

The `runRace()` method contains code that needs to be analyzed more carefully. In the first line, in fact, the method `random()` of the class `Math` is called (it belongs to the `java.lang` package that is imported automatically). The `Math` class abstracts the concept of mathematics and will be described later in this book. It contains methods that abstract mathematical functions, such as the square root or the logarithm. Among these methods we use the `random()` method which returns a randomly generated `double` type number, between `0` and `0.9999999...` (i.e. the `double` number immediately smaller than 1). In the exercise we multiplied this number by 4, obtaining a random `double` number between `0` and `3.999999999...` This is then converted to integer, so all decimal digits are truncated. We therefore obtained that the variable `winnerNumber` stores at runtime a randomly generated number, between `0` and `3`, or the possible indexes of the grid array.

The `runRace()` method then generates a random number between `0` and `3`. It uses it to identify the `Car` object of the grid array that wins the race, and then set the `result` using the `getDetails()` method of the `Car` object (which the reader will write).

All other methods of the class are accessor and mutator methods.

Exercise 6.c) Access Modifiers, `static`, and Packages, True or False:

1. A class declared as `private` cannot be used outside the package in which it is declared.

2. The following class declaration is incorrect:

```
public static class Class {...}
```

3. The following class declaration is incorrect:

```
protected class Classe {...}
```

4. The following method declaration is incorrect:

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

5. A static method can only use static variables and, to be used, it is not necessary to instantiate an object from the class in which it is defined.

6. If a method is declared `static`, it cannot be called outside of its package.

7. A `static` class is not accessible outside the package in which it is declared.

8. A `protected` method is inherited in every subclass whatever its package.

9. A `static` variable is shared by all instances of the class to which it belongs.

10. If we don't prefix modifiers to a method, the method is only accessible within the same package.

Exercise 6.d) Object Orientation in Java (Practice), True or False:

1. A `static` method must be also `public`.

2. Encapsulation implementation involves the use of the set and get keywords.

3. To use the encapsulated variables of a superclass in a subclass you must declare them at least `protected`.

4. Declared `private` methods are not inherited in subclasses.

5. An instance initializer is invoked before constructors.

6. A `private` variable is directly available (technically as if it were `public`) using the dot operator, to all instances of the class in which it is declared.

7. The keyword `this` allows you to reference the members of an object that will be created only at runtime within the object itself.

8. The keyword `this` is always optional.

9. The keyword `this` allows you to call a constructor, from a method of the same class with `this()` syntax. However, this must be the first instruction of the method.

10. The singleton pattern allows you to create a class that can be instantiated only once.

Exercise 6.e)

Abstract the concept of `Coin` with a class (complete with comments). We assume that all the coins will have the EURO as the currency, and will have the encapsulated `value` variable. Does it make sense to create a coin without specifying its value? Create a *constraint* so that coins must be instantiated with a value.

> **It is convenient to print a sentence in every important method to be able to verify the successful execution of our code. For example, when a currency is instantiated. This advice also applies to the next exercises.**

Exercise 6.f)

Considering the `Coin` class created in the previous exercise, is it correct to create the `value` variable encapsulated with the methods `setValue()` and `getValue()`? Change the class in such a way as to best abstract the class.

Exercise 6.g)

Create a `CoinsTest` class with a `main()` method that instantiates a 20 cents coin and a 1 cent coin and executes the application. Is there anything wrong with what is printed? If so, change the code so that the prints are without grammatical errors.

Exercise 6.h)

In the `CoinsTest` class you can also instantiate a 1 Euro coin. Probably there will be another bug when printing, fix it. Also add a `getDescription()` method in the `Coin` class, that returns a descriptive string of the current coin.

Exercise 6.i)

Create a class (complete with comments) `Purse` that abstracts the concept of purse. This must be able to contain a maximum of 10 coins (the `Coin` class should already have been created in the previous exercise). For now, only create a constructor that allows you to set the coins to be contained.

> **Also in this case, it is advisable to print the description of the actions being invoked.**

Exercise 6.j)

In the `CoinsTest` class, you can also instantiate a `Purse` object with 8 coins and another one with 11 coins, check that everything works correctly. Update the class comments if necessary.

Exercise 6.k)

Create an `add()` method within the class that allows you to add a coin to the purse. Provide the appropriate consistency checks.

Exercise 6.l)

Create a `state()` method that prints the current contents of the purse.

Exercise 6.m)

Create a `withdraw()` method in the class that allows you to get (and then remove) a coin from the purse. Provide the appropriate consistency checks.

Exercise 6.n)

Modify the `CoinsTest` class so as to test the created classes as completely as possible.

Exercise 6.o)

Encapsulate the `User` class of the 5.z exercise, and modify the `Authentication` class accordingly to keep everything working.

Exercise 6.p)

Draw a class diagram containing the two `User` and `Authentication` classes modified in the previous exercise that show their variables and their methods. See Appendix G containing a reference schema for UML syntax. In particular for static members (which must be underlined), and the aggregation notation that exists between the `Authentication` container class and the `User` contained class. Also use multiplicity notation. Also include the two classes in package notation.

143

Exercise 6.q)

We have already noted at the end of Exercise 5.z, that the implemented code solution did not satisfy us. In fact, during our analysis we had identified an execution flow based on different classes and methods, as shown in the sequence diagrams of the solutions of the 5.v exercise, which we report for convenience.
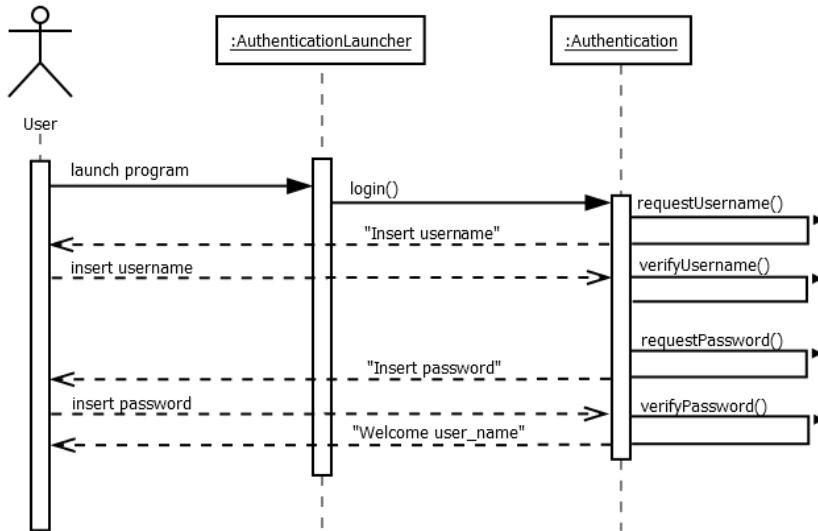
Figure 6.q.1 - (Equals to Figure 5.v.1) Sequence diagram that represents the main scenario.
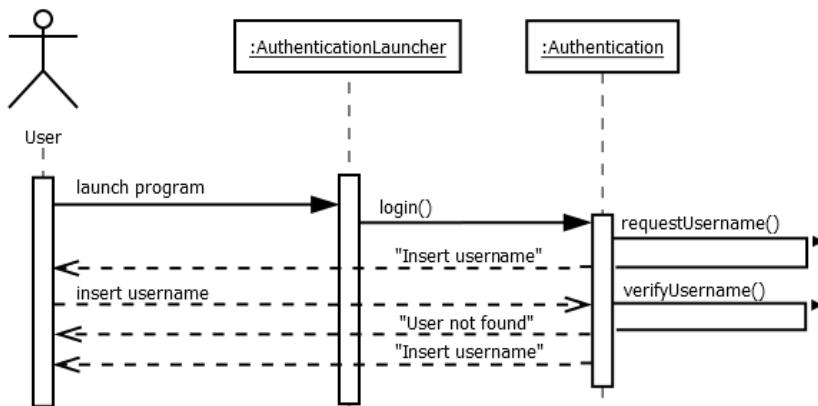
Figure 6.q.2 - (Equals to Figure 5.v.2) Sequence diagram that represents the second scenario.
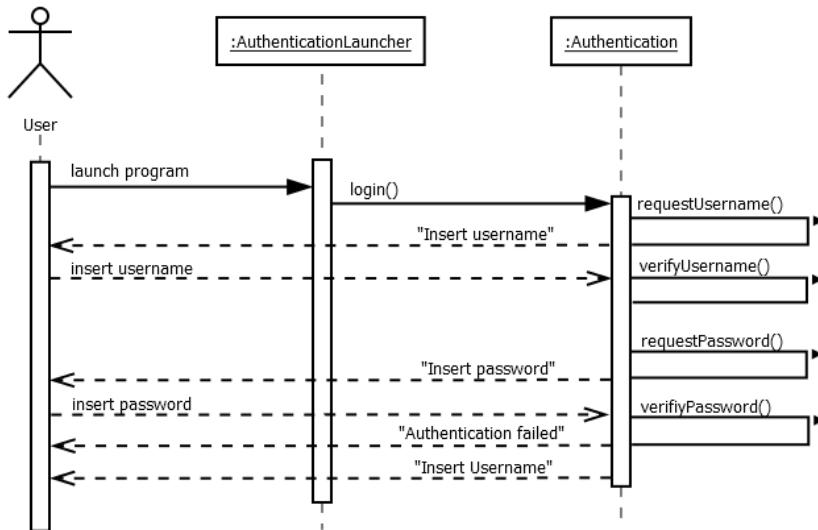
Figure 6.q.3 - (Equals to Figure 5.v.3) Sequence diagram that represents the third scenario.

Create a class diagram that correctly represents the classes needed to make the described scenarios work with the sequence diagrams, supporting themselves with the syntax reference of the Appendix G. Furthermore, transform the `Authentication` class into a singleton, as described in section 6.9.6.

It is also possible to define new methods or variables if appropriate. If you can't define the details of the class (for example return types, names or types of parameters and so on, simply don't define them).

> Obviously, the `User` class must remain encapsulated.

Exercise 6.r)

Once we implement the solution of the previous exercise, we should have an `Authentication` class with different methods and responsibilities. It has a single public method that manages the flow (`login()`) and several private methods, some verify the correctness of the data entered by the user (using the instance variable users) and others print messages to the user.

An object that is called `Authentication` is rightly responsible for:

1. Manage the application execution flow.

2. Check the correctness of the data entered by the user.

3. Contain the list of users.

4. Print messages.

Responsibilities identify the abstraction of the class, and this class is certainly too charged with responsibilities. Let's then evolve our class diagram: we try to abstract the `Authentication` class, in the most correct way and find other abstractions (classes) that can implement more specific responsibilities. Let's start by finding a class that contains the data we want to work with. What do we call it? What variables and methods should it contain? Since we have to think in an object-oriented way, let's try to make it as reusable as possible, but also consistent with the context in which we are defining it.

Exercise 6.s)

Continuing the previous exercise, modify the class diagram by identifying a class that has the responsibility to print the messages on the screen.

Exercise 6.t)

Continuing the previous exercise, is the `Authentication` class now abstract correctly? Confirm or modify the diagram again.

Exercise 6.u)

Based on the conclusion of the previous exercise, implement the code solution closest to the planned solution. Compared to the solution we had reached in exercise 5.z, we should have the same functionality, but a simpler code with which to interact, better abstract, and more reusable.

Exercise 6.v)

Given the following class:

```
public class BluRay {
    int maxGBSize = 25
    byte[] content;

    BluRay() {
    }

    void setContent(byte[] bytes) {
```

```
        this.content = content;
    }

    byte[] getContent() {
        return content
    }
}
```

add the most appropriate modifiers for each member.

Exercise 6.w)

Which of the following static import declarations are valid:

1. `import static java.lang.*;`

2. `import static java.lang.Math;`

3. `import static java.lang.Math.*;`

4. `import static java.lang.Math.PI;`

5. `import static java.lang.Math.random();`

6. `import static java.lang.Math.random;`

Exercise 6.x)

What is the output of the following program?

```
public class InitTest {

    {
        System.out.println("Initializer");
    }

    static {
        System.out.println("Static Initializer");
    }

    public InitTest () {
        System.out.println("Constructor");
    }

    public void method() {
        System.out.println("Method");
    }
```

```
    public static void staticMethod() {
        System.out.println("Static Method");
        new InitTest().method();
    }

    public static void main(String args[]) {
        InitTest.staticMethod();
    }
 }
```

Exercise 6.y)

Create an encapsulated Book class, so that it abstracts the concept of a book that can be sold in a bookstore. Among the fields defined by the Book class, there must be the genre field (understood as a literary genre). Let's make it possible for a book to be associated with only a literary genre included between a set of predefined literary genres, for example the set consisting of the genres: novel, thriller, essay, manual. Create a class that tests that Book objects work correctly.

Exercise 6.z)

Taking into account the previous exercise, create also the Bookcase class (understood as a bookcase inside a bookstore). Each bookcase must be dedicated to a certain literary genre included in the list of genres chosen in the previous exercise, and therefore must contain only books of the same genre. In the Bookcase class there must be a method called addBook(Book book), which allows you to add a book with the correct genre to the bookcase. Also create the Bookstore class. A bookstore must contain only one bookcase for each genre, and therefore it is necessary to prevent that two bookcases with the same genre can be added. In addition, the Bookstore class must implement the Singleton pattern. Finally create a BookstoreTest class, which creates an object of type Bookstore adding to it objects of type Bookcase to which have been added objects of type Book. Verify that two bookcases with the same genre cannot be added to the bookstore.

Exercise 6.aa) Static import, True or False:

Static imports allow you not to reference imported static members. After statically importing a variable, it is not possible to use an object reference to refer to it within the code.
The following import is incorrect because java.lang is always implicitly imported:

```
import static java.lang.System.out;
```

It is not possible to import nested and/or anonymous classes statically. In some cases, static

148

imports may worsen the readability of our files.

Considering the following enumeration:

```
package mypackage;
public enum MyEnum {
   A,B,C
}
```

the following code is valid:

```
import static mypackage.MyEnum.*;
public class MyClass {
  public MyClass(){
    out.println(A);
  }
}
```

If we use static imports, we could also import two static members with the same name. Their use within the code would lead to errors in compilation, if we do not use references.

Shadowing is a phenomenon that could occur if static imports are used.

Essentially the usefulness of static imports lies in the possibility of writing less code that is probably superfluous.

It makes no sense to statically import a variable if it is then used only once within the code.

It is not recommended to use static imports to import enumeration elements.

Chapter 6
Exercise Solutions

Encapsulation and Scope

1. **False**, has existed since the 1960s.

2. **True**.

3. **False**, each language provides support for the various paradigms in different ways.

4. **True**.

5. **True**.

6. **True**.

7. **True**.

8. **True**.

9. **True**.

10. **False**, you need to know the public interface and not the internal implementation.

Solution 6.b)

The code could be similar to the following:

```java
public class Driver {
    private String name;

    public Driver(String name) {
        setName(name);
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class Car {
    private String stable;
    private Driver driver;

    public Car(String stable, Driver driver) {
      setStable(stable);
      setDriver(driver);
    }

    public void setStable(String stable) {
      this.stable = stable;
    }

    public String getStable() {
      return stable;
    }

    public void setDriver(Driver driver) {
      this.driver = driver;
    }

    public Driver getDriver() {
      return driver;
    }

    public String getDetails() {
      return getDriver().getName() + " on " + getStable();
    }
  }
}
```

Solution 6.c) Modifiers and package, True or False:

1. **False**, `private` cannot be used with a class declaration.

2. **True**, `static` cannot be used with a class declaration.

3. **True**, `protected` it is not applicable to classes.

4. **True**, `static` must be positioned before the `void` keyword.

5. **True**.

6. **False**, `static` it is not an access modifier.

7. **False**, `static` it is not applicable to classes.

8. **True**.

9. **True**.

10. **True**.

Solution 6.d) Object Orientation in Java (Practice), True or False:

1. **False**.

2. **False**, these are not keywords but just a convention.

3. **False**, they can be private and be used via the accessor and mutator methods.

4. **True**.

5. **True**.

6. **True**.

7. **True**.

8. **False**, if there is ambiguity between names of instance and local variables, the keyword `this` is fundamental.

9. **False**, only another constructor of the same class can use that syntax.

10. **True**.

Solution 6.e)

The code could be similar to the following:

```
/**
 * This class abstract the concept of Coin.
 *
 * @author Claudio De Sio Cesari
 */
public class Coin {

    /**
     * The currency is a constant set to EURO.
     */
    public final static String CURRENCY = "EURO";

    /**
     * Represents the coin value in cents.
     */
    private int value;

    /**
     * Constructor that takes as input the coin value.
     *
     * @param value ithe coin value.
     */
    public Coin(int value) {
        this.value = value;
        System.out.println("Created a coin from " + value + " cents ");
    }

    /**
     * Set the value instance variable.
     *
     * @param value contains the value at which the value
     * of the  instance variable value has to be set.
     */
    public void setValue(int value) {
        this.value = value;
    }

    /**
     * Retrieves the value instance variable.
     *
     * @return
     *            the value instance variable
     */
    public int getValue() {
        return value;
    }
}
```

A constructor is sufficient to specify the required constraint. Furthermore, the currency being fixed for all the coins, has been declared as a static constant.

Solution 6.f)

In the current situation, where the specifications required only to create a class that has to always be instantiated with a value, the question could be ambiguous. However, having no other explicit constraints, it is reasonable to think of having the constraints that exist in the real world. A currency that has a specified value (let's say 5 cents) can never change its value. So, the setValue() method seems superfluous at least. So, it would be correct to remove it. It is also advisable to declare the variable final to reinforce the concept of immutability. Below the modified code:

```java
/**
 * This class abstract the concept of Coin.
 *
 * @author Claudio De Sio Cesari
 */
public class Coin {

    /**
     * The currency is a constant set to EURO.
     */
    public final static String CURRENCY ="EURO";

    /**
     * Represents the coin value in cents.
     */
    private final int value;

    /**
     * Constructor that takes as input the coin value.
     *
     * @param value ithe coin value.
     */
    public Coin(int value) {
        this.value = value;
        System.out.println("Created a  "+ value +" cents coin");
    }

    /**
     * Retrieves the value instance variable.
     *
     * @return
     *          the value instance variable
     */
```

```
    public int getValue() {
        return value;
    }
}
```

The code is more compact, but perhaps, at least for the first times, it is better to use the variables to better memorize the definitions.

Solution 6.g)

The code of the `CoinsTest` class could be the following:

```
public class CoinsTest {
    public static void main(String args[]) {
        Coin twentyCentsCoin = new Coin(20);
        Coin oneCentCoin = new Coin(1);
    }
}
```

By running this application the output will be:

```
Created a 20 cents of EURO coin
Created a 1 cents of EURO coin
```

But it would be more correct that in the second line the word "cents" was "cent".
To solve this problem, we could modify the `Coin` class in the following way (we report only the constructor responsible for printing and a utility method):

```
    public Coin(int value) {
        this.value = value;
        System.out.println("Created a "+ formatMeasurementUnit(value) +
        CURRENCY + " coin ");
    }

    private static String formatMeasurementUnit(int value) {
        return value + (value == 1 ?  " cent of " : " cents of ");
    }
```

We have delegated to a new private utility method the formatting of a piece of the sentence to be printed, using a simple ternary operator (see section 4.3.5), and we have fixed our bug. Now re-running the `CoinsTest` class we will get the following output:

```
Created a 20 cents of EURO coin
Created a 1 cent of EURO coin
```

156

Solution 6.h)

The `CoinsTest` class code should only be enriched with such an instruction:

```
Coin oneEuroCoin = new Coin(100);
```

The execution of this application will produce the following output:

```
Created a coin of 20 cents of EURO
Created a coin of 1 cent of EURO
Created a coin of 100 cents of EURO
```

But it would be more correct that in the third line "100 cents of EURO" was "1 EURO".
To solve this problem, we could modify the `Coin` class in the following way (we report only how to change the utility method):

```
private static String formatDescriptiveString(int value) {
    String formattedString =" cents of ";
    if (value == 1) {
        formattedString =" cent of ";
    } else if (value > 99) {
        formattedString =" ";
        value /= 100;
    }
    return value + formattedString;
}
```

We have modified the private utility method introduced in the previous exercise. It was not possible to use the ternary operator, so we used an `if` construct, and we fixed our bug. Now re-running the `CoinsTest` class we will get the following output:

```
Created a coin of 20 cents of EURO
Created a coin of 1 cent of EURO
Created a coin of 1 EURO
```

The `getDescription()` method could therefore be coded in this way:

```
/**
 * Retrieves the current coin description.
 *
 * @return
 *         the current coin description.
 */
public String getDescription() {
    String description ="coin of "+ formatDescriptiveString(value)
        + CURRENCY;
    return description;
}
```

157

And therefore, also the constructor could reuse this method in the following way:

```
public Coin(int value) {
    this.value = value;
    System.out.println("Created a "+ getDescription());
}
```

Solution 6.i)

The listing of the Purse class could be the following:

```
/**
 * Abstracts the concept of purse that can contain a limited number of coins.
 *
 * @author Claudio De Sio Cesari
 */
public class Purse {

    /**
     * An array that can contain a limited number of coins.
     */
    private final Coin[] coins = new Coin[10];

    /**
     * Create a Purse object containing coins whose values are
     * specified by the values varargs .
     *
     * @param values
     *          a varargs of coin values .
     */
    public Purse(int... values){
        int numberOfCoins = values.length;
        for (int i = 0; i < numberOfCoins; i++) {
            if (i >= 10) {
                System.out.println(
              "Only the first 10 coins were inserted!");
                break;
            }
            coins[i] = new Coin(values[i]);
        }
    }
}
```

Note that we have used an array of 10 Coin objects (declared final), which will act as a container for our coins. Also, we used a varargs values, to set the contents of the Purse. This will be handy when we actually create coin purses.

If more than ten values are passed to the constructor, this will only set the first ten and print a warning message.

158

Solution 6.j)

The modified `CoinsTest` class code should look something like this:

```
/**
 * Test classe for the Coin and Purse classes.
 *
 * @author Claudio De Sio Cesari
 */
public class CoinsTest {
    public static void main(String args[]) {
        Coin twentyCentsCoin = new Coin(20);
        Coin oneCentCoin = new Coin(1);
        Coin oneEuroCoin = new Coin(100);
        // Creation of a Purse with 8 coins
        Purse purse = new Purse(2, 5, 100, 10, 50, 10, 100,
            200);
        // Creation of a Purse with 11 coins
        Purse purseToFail = new Purse(2, 5, 100, 10,
            50, 10, 100, 200, 10, 5, 2);
    }
}
```

The output should be the following:

```
Created a coin of 20 cents of EURO
Created a coin of 1 cent of EURO
Created a coin of 1 EURO
Created a coin of 2 cents of EURO
Created a coin of 5 cents of EURO
Created a coin of 1 EURO
Created a coin of 10 cents of EURO
Created a coin of 50 cents of EURO
Created a coin of 10 cents of EURO
Created a coin of 1 EURO
Created a coin of 2 EURO
Created a coin of 2 cents of EURO
Created a coin of 5 cents of EURO
Created a coin of 1 EURO
Created a coin of 10 cents of EURO
Created a coin of 50 cents of EURO
Created a coin of 10 cents of EURO
Created a coin of 1 EURO
Created a coin of 2 EURO
Created a coin of 10 cents of EURO
Created a coin of 5 cents of EURO
Only the first 10 coins were inserted!
```

Solution 6.k)

The proposed code also creates as a solution a private utility method that returns the first index of the free array to contain the new currency:

```java
/**
 * Adds a coin to the purse. If this is full the coin will not
 * be added and a significant error will be printed.
 *
 * @param coin
 *           the coin to add.
 */
public void add(Coin coin) {
    System.out.println("Let's try adding one " + coin.getDescription());
    int freeIndex = firstFreeIndex();
    if (freeIndex == -1) {
        System.out.println("Purse full! The coin " +
            coin.getDescription() + " has not been added!");
    } else {
        coins[freeIndex] = coin;
        System.out.println(coin.getDescription() + " has been added");
    }
}

/**
 * Retrieves the first free index in the coin array or -1 if the
 * coin purse is full.
 *
 * @return
 *           the first free index in the coin array or -1 if the
 *           coin purse is full.
 */
private int firstFreeIndex() {
    int index = -1;
    for (int i = 0; i < 10; i++) {
        if (coins[i] == null) {
            index = i;
            break;
        }
    }
    return index;
}
```

Solution 6.l)

The code for the state() method could be the following:

```java
/**
 * Print the contents of the purse.
 */
```

```java
    public void state() {
        System.out.println("The purse contains:");
        for (Coin coin : coins) {
            if (coin == null) {
                break;
            }
            System.out.println("One " + coin.getDescription());
        }
    }
```

Solution 6.m)

The listing for the method `withdraw()` could be the following (also in this case we have created a private utility method):

```java
    /**
     * Performs a withdrawal of the specified coin from the current coin purse.
     * In case the specified currency is not present, a significant error
     * will be printed and null will be returned.
     *
     * @param coin
     *            the coin to take.
     * @return
     *            the coin found, or null if not found.
     */
    public Coin withdraw(Coin coin) {
        System.out.println("Let's try to get a "  +
            coin.getDescription());
        Coin foundCoin = null;
        int foundCoinIndex = foundCoinIndex(coin);
        if (foundCoinIndex == -1) {
            System.out.println("Coin not found!");
        } else {
            foundCoin = coin;
            coins[foundCoinIndex] = null;
            System.out.println("One " + coin.getDescription() + " withdrawn");
        }
        return foundCoin;
    }

    private int foundCoinIndex(Coin coin) {
        int foundCoinIndex = -1;
        for (int i = 0; i < 10; i++) {
            if (coins[i] == null) {
                continue;
            }
            int coinInPurseValue = coins[i].getValue();
            int valore = coin.getValue();
```

```
            if (valore == coinInPurseValue) {
                foundCoinIndex = i;
                break;
            }
        }
        return foundCoinIndex;
    }
```

Solution 6.n)

As a solution we propose a code that tries to test also the error situations:

```
/**
 * Test classe for the Coin and Purse classes.
 *
 * @author Claudio De Sio Cesari
 */
public class CoinsTest {
    public static void main(String args[]) {
        Coin twentyCentsCoin = new Coin(20);
        Coin oneCentCoin = new Coin(1);
        Coin oneEuroCoin = new Coin(100);
        // Creation of a Purse with 11 coins
        Purse purseToFail = new Purse(2, 5, 100, 10,
            50, 10, 100, 200, 10, 5, 2);
        // Creation of a Purse with 8 coins
        Purse purse = new Purse(2, 5, 100, 10, 50, 10, 100,
            200);
        purse.state();
        // we add a 20 cents coin
        purse.add(twentyCentsCoin);
        // we add a 1 cents coin
        purse.add(oneCentCoin);
        // We add the eleventh coin (we should get an error and the
        // coin will not be added)
        purse.add(oneEuroCoin);
        // We evaluate the status of the purse
        purse.state();
        // we withdraw 20 cents
        purse.withdraw(twentyCentsCoin);
        //Let's add the tenth coin again
        purse.add(oneEuroCoin);
        // We evaluate the status of the purse
        purse.state();
        // We withdraw a non-existent currency (we should get an error)
         purse.withdraw(new Coin(7));
    }
}
```

The output should be the following:

```
Created a coin of 20 cents of EURO
Created a coin of 1 cent of EURO
Created a coin of 1 EURO
Created a coin of 2 cents of EURO
Created a coin of 5 cents of EURO
Created a coin of 1 EURO
Created a coin of 10 cents of EURO
Created a coin of 50 cents of EURO
Created a coin of 10 cents of EURO
Created a coin of 1 EURO
Created a coin of 2 EURO
Created a coin of 10 cents of EURO
Created a coin of 5 cents of EURO
Only the first 10 coins were included!
Created a coin of 2 cents of EURO
Created a coin of 5 cents of EURO
Created a coin of 1 EURO
Created a coin of 10 cents of EURO
Created a coin of 50 cents of EURO
Created a coin of 10 cents of EURO
Created a coin of 1 EURO
Created a coin of 2 EURO
The purse contains:
One coin of 2 cents of EURO
One coin of 5 cents of EURO
One coin of 1 EURO
One coin of 10 cents of EURO
One coin of 50 cents of EURO
One coin of 10 cents of EURO
One coin of 1 EURO
One coin of 2 EURO
Let's try adding one coin of 20 cents of EURO
coin of 20 cents of EURO has been added
Let's try adding one coin of 1 cent of EURO
coin of 1 cent of EURO has been added
Let's try adding one coin of 1 EURO
Purse full! The coin coin of 1 EURO has not been added!
The purse contains:
One coin of 2 cents of EURO
One coin of 5 cents of EURO
One coin of 1 EURO
One coin of 10 cents of EURO
One coin of 50 cents of EURO
One coin of 10 cents of EURO
One coin of 1 EURO
One coin of 2 EURO
One coin of 20 cents of EURO
One coin of 1 cent of EURO
Let's try to get a coin of 20 cents of EURO
One coin of 20 cents of EURO withdraw
Let's try adding one coin of 1 EURO
coin of 1 EURO has been added
The purse contains:
```

```
One coin of 2 cents of EURO
One coin of 5 cents of EURO
One coin of 1 EURO
One coin of 10 cents of EURO
One coin of 50 cents of EURO
One coin of 10 cents of EURO
One coin of 1 EURO
One coin of 2 EURO
One coin of 1 EURO
One coin of 1 cent of EURO
Created a coin of 7 cents of EURO
Let's try to get a coin of 7 cents of EURO
Coin not found!
```

Solution 6.o)

The encapsulated User code is as follows:

```java
package com.claudiodesio.authentication;

public class User {
    private String name;
    private String username;
    private String password;

    public User(String name, String username, String password) {
        this.name = name;
        this.username =username;
        this.password =password;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }
```

164

```
    public void setUsername(String username) {
        this.username = username;
    }
}
```

while `Authentication` changes little: it is only necessary to replace direct access to `User` public variables, with the corresponding calls to accessor methods:

```
package com.claudiodesio.authentication;
import java.util.Scanner;

public class Authentication {

    private static final User[] users = {
        new User("Daniele", "dansap", "music"),
        new User("Giovanni", "giobat", "science"),
        new User("Ligeia", "ligder", "art")
    };

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("Type  username.");
            String username = scanner.nextLine();
            User user = verifyUsername(username);
            if (user == null) {
                System.out.println("User not found!");
                continue;
            }
            System.out.println("Type password");
            String password = scanner.nextLine();
            if (password != null && password.equals(user.getPassword())) {
                System.out.println("Hello " + user.getName());
                break;
            } else {
                System.out.println("Authentication failed");
            }
        }
    }

    private static User verifyUsername(String username) {
        if (username != null) {
            for (User user : users) {
                if (username.equals(user.getUsername())) {
                    return user;
                }
            }
        }
        return null;
    }
}
```

Solution 6.p)

Figure 6.p.1 shows the required class diagram. Notice how the members of the `Authentication` class are marked `static` with an underline. Furthermore, the aggregation (which indicates the containment relation), is addressed by the object contained to the containing object, and is distinguished syntactically from a simple association (*relation of use*) by drawing a white diamond on the side of the contained object. The asterisk * symbol next to the object contained instead, describes the multiplicity of the contained object. Finally, note that on the side of the containing object there are no multiplicity symbols, this means that it is as if the multiplicity of default were present, i.e. 1. In fact, an `Authentication` object contains multiple `User` objects.

```
com.claudiodesio.authentication

               Authentication
        -users: User[]
        +verifyUsername(username:String): User
        +main(args[]:String): void

                      △
                      △
                      │
                      ◇  *

                     User
        -username: String
        -password: String
        -name: String
        +User(name:String,username:String,password:String)
        +setName(name:String): void
        +getName(): String
        +setUsername(username:String): void
        +getUsername(): String
        +setPassword(password:String): void
        +getPassword(): String
```

Figure 6.p.1 - Class diagram of the com.claudiodesio.autentication package.
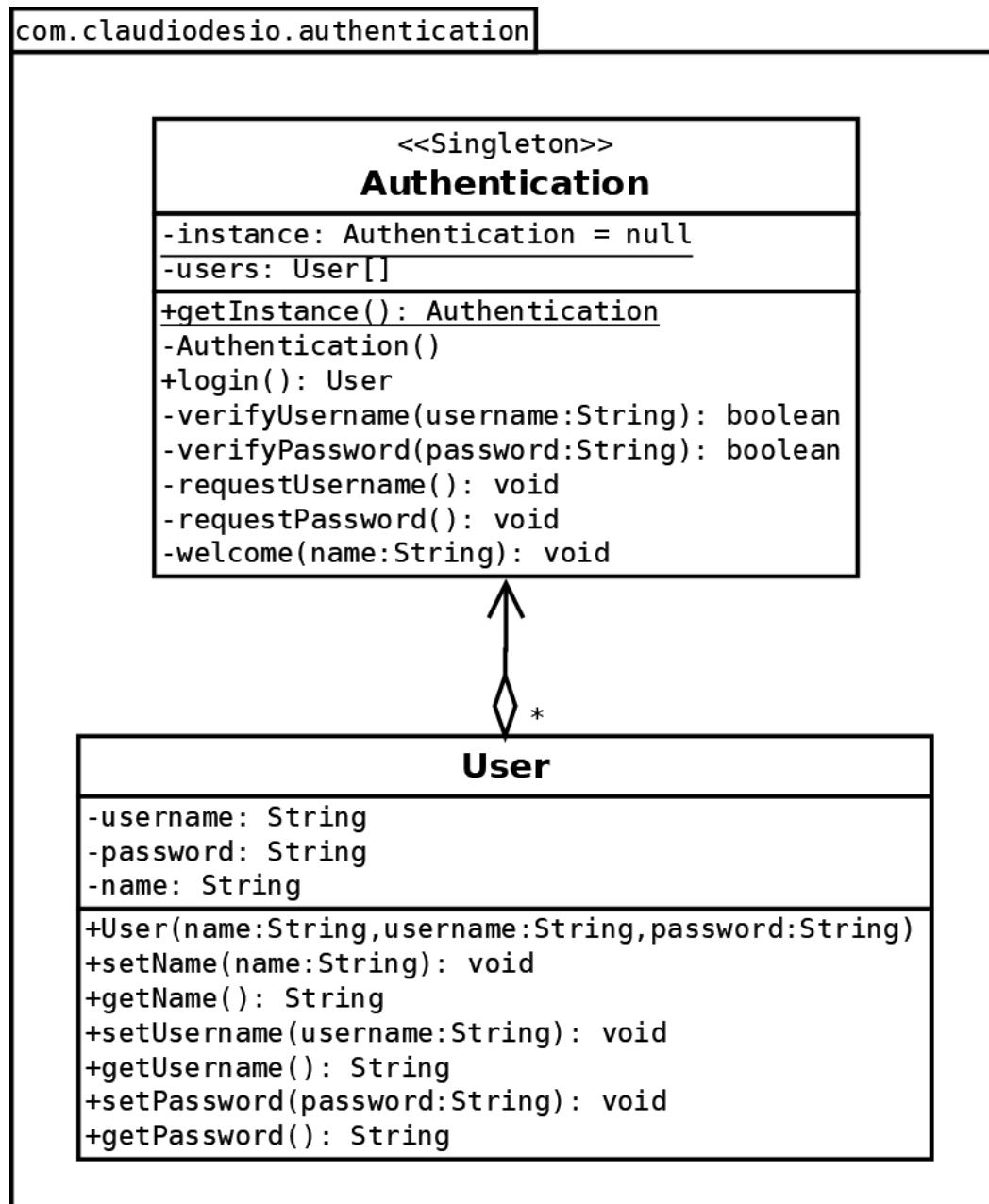
Figure 6.q.1 - Class diagram of the com.claudiodesio package. Authentication modified as required.

Solution 6.q)

Figure 6.q.1 shows the required class diagram. Note that to transform the class into a singleton, we have defined a private constructor, a static variable of type of `Authentication`, and a `getInstance()` method that has the responsibility to always return the same instance of `Authentication`, or `instance` variable, which is appropriately instantiated only once.

Since the class is now a singleton and will be instantiated, we have made sure that its methods and its variables are no longer `static`. We have written all the methods that have been described in the sequence diagrams and we have added the return types and the arguments in our opinion more correct. Actually, we will find out if these are really correct when we deal with the implementation, this is only our idea for now, and also a superficial idea (think of the code solution we implemented in the exercise 5.z, where the result was really different from what we expected).

In particular we have intended the `requestUsername()` and `requestPassword()` methods as printing methods. In fact, they do not take input parameters or even declare return types. The `verifyUsername()` and `verificationPassword()` methods should instead return a boolean (`true` if the check is successful and `false` if it fails). We have also added the methods `welcome()`, `authenticationFailed()`, and `usernameNotFound()` intended as printing methods, even if they have not been reported in the sequence diagrams (the names are self-explanatory). All these methods are private methods, while the only public method is the `login()` method, which manages the flow of calls to private methods.

In our mind the `Authentication` class must work like this (for now!).

Solution 6.r)

As you can see in Figure 6.r.1, we have created a new class called `UserProfiles`, which acts as a database and contains information about users (the `user` array). We have decided that an instance of this class will replace the `users` array that previously resided within the `Authentication` class, in order not to lose the `Authentication` class information about users. Now the classes are better abstracted, as each has a specific role. We realized that the singleton design pattern makes more sense than it is implemented in the `UserProfiles` class, compared to in `Authentication`. In fact, it is the data that must be unique for all classes, not the authentication process. So, we acted accordingly.
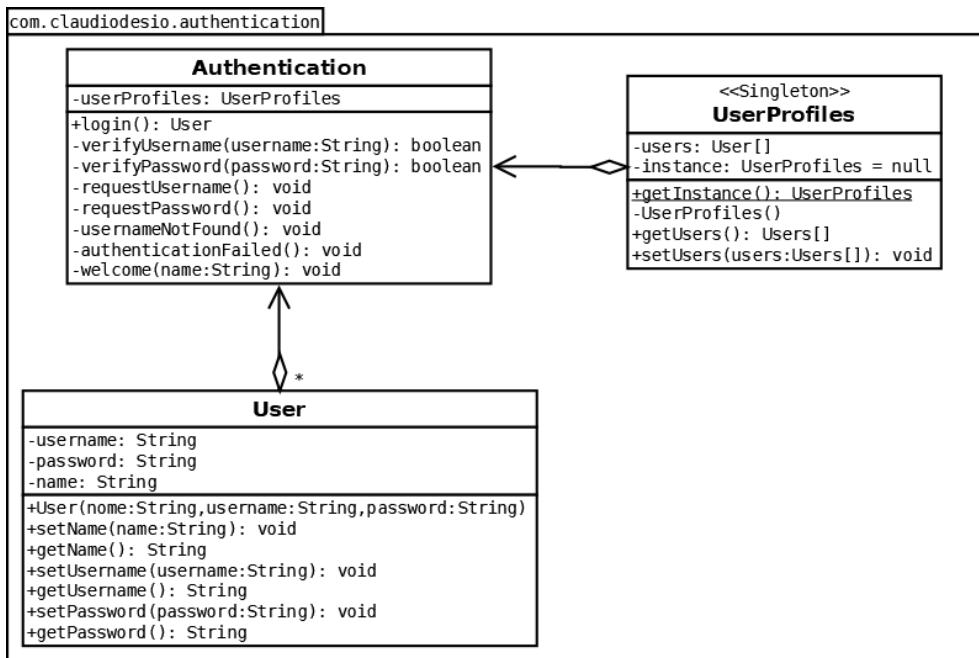
Figure 6.r.1 - Class diagram of the com.claudiodesio package. Authentication with UserProfiles.
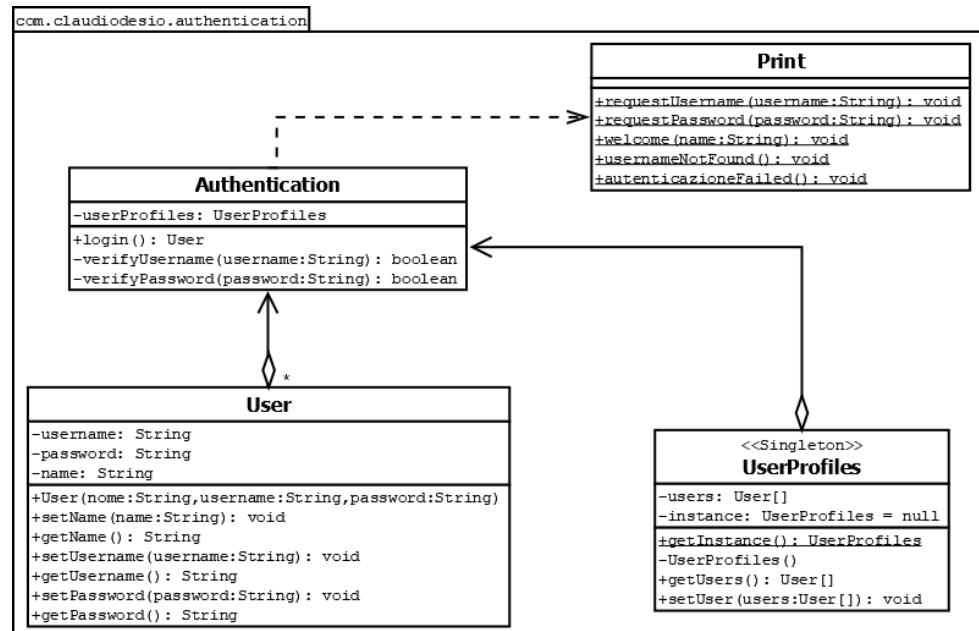
Figure 6.s.1 - Class diagram of the com.claudiodesio package. Authentication with Print.

Solution 6.s)

We have created a simple utility class called `Print`, containing all methods that send messages to the user of the application. We made them all static, because it seems superfluous to instantiate a class that contains only printing methods without defining instance variables.

> **The latter is a choice like any other, not necessarily the best one. Declaring static methods implies ignoring the advantages of extensibility that we will see in the next chapters, but our choice is not to be condemned.**

Solution 6.t)

Is the class correctly abstracted? It depends on your point of view! The `Authentication` class is responsible for defining the login flow and verifying the correctness of the data. How it is, it seems fine. However, one could also think of delegating the verification of the correctness of the data to another class of utility that we could call `Verifier`. This class could contain the two methods of verification declared `static`, or it could contain a constructor to which we pass the instance of `User` that we want to verify. They are all valid choices, each of which has consequences. Not creating the `Verifier` class would imply having a bigger `Authentication` class, but creating it would imply an extra class (among other things strictly dependent on the `User` class). For now, we opt to leave things as they are. We will decide later when we have a clearer picture.

Solution 6.u)

As we have said, the `User` class remains unchanged:

```
package com.claudiodesio.authentication;

public class User {
    private String name;
    private String username;
    private String password;

    public User(String name, String username, String password) {
        this.name = name;
        this.username =username;
        this.password =password;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

The UserProfiles class has been faithfully implemented with respect to how it was designed:

```
package com.claudiodesio.authentication;

public class UserProfiles {

    private static UserProfiles instance;

    private User[] users;

    private UserProfiles() {
        users = createUsers();
    }

    public static UserProfiles getInstance() {
        if (instance == null) {
            instance = new UserProfiles();
        }
        return instance;
    }

    private User[] createUsers() {
        User[] users = {
            new User("Daniele", "dansap", "music"),
```

```
            new User("Giovanni", "giobat", "science"),
            new User("Ligeia", "ligder", "art")
        };
        return users;
    }

    public void setUsers(User[] users) {
        this.setUsers(users);
    }

    public User[] getUsers() {
        return users;
    }
}
```

Even the `Print` class is faithful to how it was designed, except for the introduction of an extra private method: `printMessage()`, which centralizes the printing instruction. This can be useful in the future if we want to change the way we print a message, because we will have to do it only in that method and not in all the others:

```
package com.claudiodesio.authentication;

public class Print {

    public static void requestUsername() {
        printMessage("Type username.");
    }

    public static void requestPassword() {
        printMessage("Type password.");
    }

    public static void sayHello(String nome) {
        printMessage("Hello " + nome);
    }

    public static void usernameNotFound() {
        printMessage("User not found!");
    }

    public static void authenticationFailed() {
        printMessage("Authentication failed");
    }

    private static void printMessage(String message) {
        System.out.println(message);
    }
}
```

172

The Authentication class instead, has been changed:

```java
package com.claudiodesio.authentication;

import java.util.Scanner;

public class Authentication {

    public void login() {
        boolean authorized = false;
        Scanner scanner = new Scanner(System.in);
        do {
            Print.requestUsername();
            String username = scanner.nextLine();
            User user = findUser(username);
            if (user != null) {
                Print.requestPassword();
                String password = scanner.nextLine();
                if (verifyPassword(user, password)) {
                    Print.sayHello(user.getName());
                    authorized = true;
                } else {
                    Print.authenticationFailed();
                }
            } else {
                Print.usernameNotFound();
            }
        } while (!authorized);
    }

    private User findUser(String username) {
        User[] users = UserProfiles.getInstance().getUsers();
        if (username != null) {
            for (User user : users) {
                if (username.equals(user.getUsername())) {
                    return user;
                }
            }
        }
        return null;
    }
//     private boolean verifyUsername(String username) {
//         User[] users = UserProfiles.getInstance().getUsers();
//         boolean found = false;
//         User user = findUser(username);
//         if (user != null && username.equals(user.getUsername())) {
//             found = true;
//         }
//         return found;
//     }
```

```
    private boolean verifyPassword(User user, String password) {
        boolean found = false;
        if (password != null) {
            if (password.equals(user.getPassword())) {
                found = true;
            }
        }
        return found;
    }

    public static void main(String args[]) {
        Authentication authentication = new Authentication();
        authentication.login();
    }
}
```

In particular, the verifyUsername() method, which returns a boolean as we had planned, has been commented out and replaced with the findUser() method, which directly returns the User object corresponding to the username specified as a argument. If no user is found, the method returns null. This substitution allows us not to duplicate code (either to find a user with findUser()). In fact, to verify the username with verifyUsername(), we would have done the same loop, and the code of the two methods would have been almost identical.

The main() method was introduced only as a method to test the login functionality. It could be placed in any class, such as AuthenticationLauncher, which we initially identified in our analysis among the solutions of the exercises of the fifth chapter.

The login() method now contains the so-called "business logic", which is the code that satisfies the requirements thanks to an appropriate algorithm. With respect to the solution of the 5.z exercise, note the elimination of the break and continue constructs, replaced by the more convenient do-while loop, supported by the authorized boolean variable, which is set to true only when the verification procedures of the username and password are both verified. The algorithm is clearer and more linear, also thanks to the support of the Print class and the Scanner object, which are used several times to print output messages, and collect input from the application user. However, in our opinion, improvements to the algorithm and abstraction of the class can still be made. In fact, we had to improvise our solution, because the one designed with the class diagram did not prove to be worthy of being implemented. What has been missed?

The scenarios were not redesigned with interaction diagrams, after the new classes and the new methods have been identified. In these interaction diagrams, we could also specify details like parameter types, object names and return types. In fact, the first sequence diagrams we created in the 5.v exercise were based only on key abstraction, and were used to verify what to do (they were *analysis diagrams*). The diagrams that we could have created after the changes made to the

174

class diagram instead, should have been considered *design diagrams* that explained how to do it. For now, a step forward thanks to the class diagram we did it, later we will try to make others.

Solution 6.v)

This should be the solution:

```java
public class BluRay {
    public final static int maxGBSize = 25;
    private byte[] content;

    public BluRay() {
    }

    public void setContent(byte[] bytes) {
        this.content = content;
    }

    public byte[] getContent() {
        return content;
    }
}
```

Solution 6.w)

The correct statements are 3, 4 and 6.

Solution 6.x)

The program output is:

```
Static Initializer
Static Method
Initializer
Constructor
Method
```

Solution 6.y)

The required Book class could be the following:

```java
public class Book {
    private String isbn;
    private String title;
    private String author;
    private int price;
```

```java
    private String genre;
    public static final String [] genres = {"Romance", "Essay", "Thriller",
      "Handbook"};

    public Book(String isbn, String title, String author, int price,
      String genre) {
        setIsbn(isbn);
        setTitle(title);
        setAuthor(author);
        setPrice(price);
        setGenre(genre);
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String title) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
```

```
        for (String validGenre : genres) {
            if (validGenre.equals(genre)) {
                this.genre = genre;
                return;
            }
        }
        System.out.println("Genre " + genre +
            " not valid! Please, use one of the following genres:");
        for (String validGenre : genres) {
            System.out.println(validGenre);
        }
    }
}
```

Note that we have created an array of strings called genres as a static constant to define the default valid genres. We used the valid genres to test the validity of the call from the setGenere() method. Note that the use of the return command used within the method, causes its immediate termination. The return command is not followed by any value or variable because the method has void return type.

> **Note that the check defined within the setGenere() method allows to avoid the setting of the genre field in case the method parameter is not valid, but the Book type object is however created with the genre variable set to null. In Chapter 9 we will see how to handle this type of situation by launching an exception, thus avoiding instantiating a Book object with a null genre field.**

With the following class we tested the Book class:

```
public class BookTest {
    public static void main(String[] args) {
        Book jfaVol1 = new Book("979-12-200-4915-3", "Java for Aliens Vol. 1",
            "Claudio De Sio Cesari", 25, "Handbook");
        Book jfaVol2 = new Book("979-12-200-4916-0", "Java for Aliens Vol. 2",
            "Claudio De Sio Cesari", 25, "Biography");
        System.out.println("JFA Vol 1 Genre = " + jfaVol1.getGenre());
        System.out.println("JFA Vol 2 Genre = " + jfaVol2.getGenre());
    }
}
```

The output of the previous class follows:

```
Genre Biography not valid! Please, use one of the following genres:
Romance
```

177

```
Essay
Thriller
Handbook
JFA Vol 1 Genre = Handbook
JFA Vol 2 Genre = null
```

Solution 6.z)

Before defining the Bookcase class, we found it necessary to redefine the Book class in the following way:

```
public class Book {
    private String isbn;
    private String title;
    private String author;
    private int price;
    private String genre;

    public Book(String isbn, String title, String author, int price,
            String genre) {
        setIsbn(isbn);
        setTitle(title);
        setAuthor(author);
        setPrice(price);
        setGenre(genre);
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String title) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getAuthor() {
        return author;
    }
```

```
    public void setAuthor(String author) {
        this.author = author;
    }
    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        if (GenreUtils.isValidGenre(genre)) {
            this.genre = genre;
        } else {
            GenreUtils.printError(genre);
        }
    }
}
```

We can see that we have simplified the implementation, eliminating the genres array, given that since we will have to use the concept of genre also for the Bookcase class, we preferred to create a utility class that we have called GenreUtils, whose static methods we also used for simplify the implementation of the setGenere() method checks. The following is the GenreUtils class:

```
public class GenreUtils {
    public static final String ROMANCE = "Romance";
    public static final String ESSAY = "Essay";
    public static final String THRILLER = "Thriller";
    public static final String HANDBOOK = "Handbook";
    public static final String SCIFI = "Scifi";
    public static final String[] genres = { ROMANCE, ESSAY, THRILLER, HANDBOOK,
        SCIFI };

    public static boolean isValidGenre(String genre) {
        boolean validGenre = false;
        for (String fixedGenre : genres) {
            if (fixedGenre.equals(genre)) {
                validGenre = true;
            }
        }
        return validGenre;
    }
```

```
    public static void printError(String genre) {
        System.out.println("Genre " + genre +
            " not valid! use one of the followinggenres:");
        for (String fixedGenre : genres) {
            System.out.println(fixedGenre);
        }
    }
}
```

In the GenreUtils class we have reported the genres in the form of static and public constants, which if used instead of strings, make it possible to prevent typing errors from becoming bugs. We took the opportunity to add a new genre: SCIFI. In addition, for convenience we have also used them to fill the genre array, so as to use loops to iterate the elements of the array. Finally, we have declared two static and public methods. The isValidGenre() method returns true only if the genre is valid. The printError() method prints an error message. Both of these methods were used in the setGenere() method of the Book class.

So, we have defined the Bookcase class it in the following way:

```
public class Bookcase {
    private Book[] books;
    private String genre;

    public Bookcase(String genre) {
        books = new Book[100];
        setGenre(genre);
    }

    public void addBook(Book book) {
        if (genre == null) {
            System.out.println("The genre of this bookcase is still not set"
                +" and books cannot be added!");
            GenreUtils.printError(null);
            return;
        }
        for (int i = 0; i < books.length; i++) {
            if (books[i] == null) {
                books[i] = book;
                return;
            }
        }
        System.out.println("The bookcase is full!");
    }

    public void setBooks(Book[] books) {

    }
```

```java
    public Book[] getBooks() {
        return books;
    }

    public void setGenre(String genre) {
        if (GenreUtils.isValidGenre(genre)) {
            this.genre = genre;
        } else {
            GenreUtils.printError(genre);
        }
    }

    public String getGenre() {
        return genre;
    }
}
```

Note that the setGenre() method is practically the same method defined for the Book class, as it can take advantage of GenreUtils methods. The method addBook() instead, first of all checks if the current bookstore object has the genre variable initialized. If not, it prints an error message and does not perform the rest of the method. Otherwise look for the first free position in the genre array, if it doesn't find it, then print an error message to warn that there is no position available on the bookcase for other books.

Finally, the singleton class Bookstore could be the following

```java
public class Bookstore {
    private static Bookstore instance;
    private String name;
    private Bookcase[] bookcases;

    public Bookstore() {
        bookcases = new Bookcase[GenreUtils.genres.length];
    }

    public static Bookstore getInstance() {
        if (instance == null) {
            instance = new Bookstore();
        }
        return instance;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void addBookcase(Bookcase bookcase) {
```

```
            if (bookcases[bookcases.length-1] != null) {
                System.out.println("This bookstore already has all the bookcases!");
                return;
            }

            for (int i = 0; i < bookcases.length; i++) {
                if (bookcases[i] == null) {
                    bookcases[i] = bookcase;
                    break;
                } else if (bookcases[i].getGenre().equals(bookcase.getGenre())) {
                    System.out.println("The "+ bookcase.getGenre()
                        +" bookcase already exists!");
                    break;
                }
            }
        }

        public Bookcase[] getBookcases() {
            return bookcases;
        }
    }
```

We can see that within this class we have always used the `length` variable of the arrays involved, so the code will not change in the case the number of genres will change.

The `addBookcase()` method instead, implements an algorithm that first checks if all the bookcases have already been set. Then within a loop that iterates the bookcases, it looks for the first free position in the array. For each bookcase already within the array, the method checks whether it has a `genre` coinciding with that of the bookcase that was passed as a parameter (if so, prints an error message and ends the method with a `return` command) .

We can test everything with the following class:

```
public class BookstoreTest {
    public static void main(String[] args) {
        Book jfaVol1 = new Book("979-12-200-4915-3", "Java for Aliens Vol. 1",
            "Claudio De Sio Cesari", 25, GenreUtils.HANDBOOK);
        Book jfaVol2 = new Book("979-12-200-4916-0", "Java for Aliens Vol. 2",
            "Claudio De Sio Cesari", 25, GenreUtils.HANDBOOK);
        Book f451 = new Book("978-88-046-6529-8", "Fahrenheit 451",
            "Ray Bradbury", 10, GenreUtils.SCIFI);
        Book shining = new Book("978-88-452-9530-0", "Shining", "Stephen King",
            12, GenreUtils.THRILLER);
        Book tkr = new Book("978-88-683-6730-5", " The Kite Runner ",
            "Khaled Hosseini", 11, GenreUtils.ROMANCE);
        Book ttoe = new Book("978-88-170-7976-1", "The Theory of Everything",
            "Stephen Hawking", 10, GenreUtils.ESSAY);
        Bookcase handbookBookcase = new Bookcase(GenreUtils.HANDBOOK);
        Bookcase scifiBookcase = new Bookcase(GenreUtils.SCIFI);
        Bookcase scifiBookcase2 = new Bookcase(GenreUtils.SCIFI);
```

```
            Bookcase thrillerBookcase = new Bookcase(GenreUtils.THRILLER);
            Bookcase romanceBookcase = new Bookcase(GenreUtils.ROMANCE);
            Bookcase essaysBookcase = new Bookcase(GenreUtils.ESSAY);
            Bookcase essaysBookcase2 = new Bookcase(GenreUtils.ESSAY);
            handbookBookcase.addBook(jfaVol1);
            handbookBookcase.addBook(jfaVol2);
            scifiBookcase.addBook(f451);
            thrillerBookcase.addBook(shining);
            romanceBookcase.addBook(tkr);
            essaysBookcase.addBook(ttoe);
            Bookstore bookstore = Bookstore.getInstance();
            bookstore.setName("Bookstore for aliens");
            bookstore.addBookcase(handbookBookcase);
            bookstore.addBookcase(scifiBookcase);
            bookstore.addBookcase(scifiBookcase2);
            bookstore.addBookcase(thrillerBookcase);
            bookstore.addBookcase(romanceBookcase);
            bookstore.addBookcase(essaysBookcase);
            bookstore.addBookcase(essaysBookcase2);
            Bookcase[] bookcases = bookstore.getBookcases();
            System.out.println("Bookstore list of bookcases:");
            for (Bookcase bookcase : bookcases) {
                System.out.println("Bookcase " + bookcase.getGenre() + ":");
                Book[] books = bookcase.getBooks();
                for (Book book : books) {
                    if (book != null) {
                        System.out.println("\t" + book.getTitle() + " by " +
                            book.getAuthor() + " (Genre " + book.getGenre() + ")");
                    }
                }
            }
        }
    }
```

whose output will be:

```
The Scifi bookcase already exists!
This bookstore already has all the bookcases!
Bookstore list of bookcases:
Bookcase Handbook:
        Java for Aliens Vol. 1 by Claudio De Sio Cesari (Genre Handbook)
        Java for Aliens Vol. 2 by Claudio De Sio Cesari (Genre Handbook)
Bookcase Scifi:
        Fahrenheit 451 by Ray Bradbury (Genre Scifi)
Bookcase Thriller:
        Shining by Stephen King (Genre Thriller)
Bookcase Romance:
        The Kite Runner by Khaled Hosseini (Genre Romance)
Bookcase Essay:
        The Theory of Everything by Stephen Hawking (Genre Essay)
```

Solution 6.aa) Static import, True o False:

1. **True**.

2. **False**.

3. **False**.

4. **True**.

5. **False**, out is not statically imported.

6. **True**.

7. **True**.

8. **True**.

9. **True**.

10. **False**.

Chapter 7
Exercises

Inheritance and Interfaces

For this chapter we will avoid having the reader write too much code. Instead, it is very important to focus rather on definitions. If you do not know all the concepts of the theory well, you will end up writing incoherent code from the point of view of the philosophy of objects

> **After each exercise look at the solution, because each one could be preparatory to the next.**

Exercise 7.a) Object Orientation in Java (Theory), True or False:

1. The implementation of inheritance always involves writing a few less lines.

2. The following class declaration is incorrect:

   ```
   public final class Class extends OtherClass {...}
   ```

3. Inheritance is only useful if specialization is used. In fact, specializing we inherit in the subclass (or subclasses) members of the superclass that we must not rewrite. Instead with the generalization we create an extra class, and then we write more code.

4. The super keyword allows you to call superclass methods and constructors. The keyword this allows you to call methods and constructors of the same class.

5. Multiple inheritance does not exist in Java because it does not exist in reality.

6. A functional interface is an interface that declares a single default method.

7. A subclass is "bigger" than a superclass (in the sense that it usually adds new features and functionality compared to the superclass).

8. Suppose we develop an application to manage a soccer tournament. There is inheritance derived from specialization between the `Team` and `Player` classes.

9. Suppose we develop an application to manage a soccer tournament. There can be inheritance derived from generalization between the `Team` and `Player` classes.

10. In general, if we had two classes, `Father` and `Son`, there would be no inheritance between these two classes

---

Exercise 7.b)

Given the following class:

```java
public class Person {
    private String name;

    public void setName(String name) {
      this.name = name;
    }

    public String getName() {
      return name;
    }
}
```

Add comments on the following `Employee` class to highlight where object-oriented paradigms are used: encapsulation, inheritance and reuse

```java
public class Employee extends Person {
    private int id;

    public void setData(String name, int id) {
      setName(name);
      setId(id);
    }

    public void setId(int id) {
        this.id = id;
    }
```

```
    public int getId() {
      return id;
    }

    public String getData() {
        return getName() +", id: "+ getId();
    }
}
```

Exercise 7.c) Abstract Classes and Interfaces, True or False:

1. The following class declaration is incorrect:

```
public abstract final class Class {...}
```

2. The following class declaration is incorrect:

```
public abstract class Class;
```

3. The following interface declaration is incorrect:

```
public final interface Class {...}
```

4. An abstract class necessarily contains abstract methods.

5. An interface can be extended by another interface.

6. A class can extend a single class but implement multiple interfaces.

7. The advantage of abstract classes and interfaces is that they force subclasses to implement inherited abstract methods. Therefore, they represent an excellent tool for object-oriented design.

8. An interface can declare more than one constructor.

9. An interface cannot declare variables but static and public constants.

10. An abstract class can implement an interface

Exercise 7.d)

Describe all inheritance relationships between the following classes:

1. Teacher

2. Student

3. `Person`

4. `Desk`

5. `Course`

6. `Classroom`

7. `Lesson`

Exercise 7.e)

If we wanted to create the hierarchy defined in the previous exercise, between `Student`, `Person` and `Teacher`, which could be an abstract class?

Exercise 7.f)

Create the interface (with comments) `Musical` declaring a method named `play()`. How would you declare this method: `static`, `default` or `abstract`?

Exercise 7.g)

Create two subinterfaces (with comments) of `Musical`: `MusicalInstrument` and `Ringtone`. How would you declare the method `play()` in the two subinterfaces: `static`, `default` or `abstract`?

Exercise 7.h)

Suppose we create a `Smartphone` class that implements both interfaces from the previous exercise. What's wrong?

Exercise 7.i) Interfaces after Java 8, True or False:

1. Static methods cannot be inherited.

2. The following interface statement is incorrect:

```
public static interface Interface;
```

3. The following interface statement is incorrect:

```
public interface Interface {}
```

4. The abstract methods of an interface are not inherited by another interface.

5. The abstract methods of an interface cannot be implemented by another interface.

6. Static methods of an interface cannot be implemented by another interface.

7. An A interface defines a default method m(). Interface B extends interface A and redefines the m() method with a default implementation. An abstract class C implements interface A but redefining the method m(). A concrete (non-abstract) class D extends the class C and implements interface B, without redefining the method m(). Class D inherits the m() method defined in class C.

8. An E interface defines an abstract method m(). The F interface extends the E interface and redefines the m() method with a default implementation. An abstract class G implements the interface E but does not redefine the method m(). A concrete (non-abstract) class H extends the abstract class G and implements the interface F, without redefining the method m(). Class H cannot be compiled correctly.

9. An interface can extend multiple interfaces.

10. An interface can extend an abstract class and an interface.

Exercise 7.j)

Resuming the solution of the Exercise 6.z, let's suppose we want to make our bookstore sell also music albums. So, let's abstract the Album class, bearing in mind that even for music albums there is an identification number called ISMN (although there are other ways to identify an album as using the European EAN standard), as well as for books there is the ISBN identification number. Then check if there is an inheritance relationship between the Book class and the Album class. If it exists, implement a solution.

> **For now, do not implement the checks that the setGenre() method of the Book class executes so that the genre specified as a parameter belonged to a predefined set of literary genres (see Exercise 6.z), since this will be fixed in the next exercise.**

Exercise 7.k)

Starting from the solutions of the previous exercise and of the Exercise 6.z, rename the GenreUtils class as LiteraryGenreUtils and create the equivalent MusicalGenreUtils for the Genre class. Check if there is an inheritance relationship between

the two classes, and if it exists, implement it. Also, the Album class must have a genre value belonging to a predefined set of musical genres (see setGenere() method of the Book class). Finally create an ItemsTest class, equivalent to the BookTest class of the Exercise 6.y, which tests the objects instantiated by the Book and Album classes.

Exercise 7.l)

Keeping in mind all the code insertions that the compiler implicitly executes, rewrite the following class, adding all the instructions that the compiler would add:

```java
public class CompilerItsYourTurn extends Object {

    private int var;

    public void setVar(int v) {
      var = v;
    }

    public int getVar() {
      return var;
    }
}
```

Exercise 7.m)

Considering the following classes:

```java
public class Person {
    private String name;

    public void setName(String name) {
      this.name = name;
    }

    public String getName() {
      return this.name;
    }
}

public class Employee extends Person {
    private int id;

    public void setId(int id) {
        this.id = id; //encapsulation
    }
```

```
    public int getId() {
      return this.id; //encapsulation
    }

    public String getData() {
        return getName() +", id: "+ getId();
    }
}
```

What will be the output of the compilation process (choose only one option)?

1. No output (correct compilation).

2. Error in the getData() of Employee method.

3. Error in the getName() of Person method.

4. Error in the getId() of Employee method.

If we add the following method to the Employee class described in the Exercise 7.m:

```
public void setData(String name, int id) {
    setName(name);
    setId(id);
}
```

What will be the output of the compilation process (choose only one option)?

1. No output (correct compilation).

2. Error in the getData() of Employee method.

3. Error in the setName() of Person method.

4. Error in the setId() of Employee method

Which of these statements is true (they could all be true)?

1. Inheritance allows you to link multiple classes together.

2. Inheritance allows you to link more interfaces to one another.

3. Inheritance allows you to link multiple classes and interfaces.

4. Inheritance makes it possible to link several classes and interfaces

Exercise 7.p)

Given the following code:

1. `class Animal {}`

2. `interface Feline {}`

3. `class Lion {}`

If we wanted to link the previous types with inheritance, which of the following snippets is valid from the compiler's point of view (they could all be valid)?

1. `class Animal extends Feline {}`

2. `interface Feline extends Animal {}`

3. `class Lion extends Feline {}`

4. `class Lion extends Animal implements Feline {}`

5. `class Animal extends Lion implements Feline {}`

Exercise 7.q)

Given that an interface can declare in addition to abstract methods, also public and private static methods, public and private implemented (`default`) methods, and given that with interfaces we can implement multiple inheritance, why should we prefer an abstract class to an interface?

Exercise 7.r)

Given the following types:

- `interface Flying {}`
- `class Plane implements Flying {}`

which of the following snippets are correct?

1. `Plane a = new Plane();`

2. `Flying v = new Flying();`

3. `plane1.equals(plane2);` (where `plane1` and `plane2` are objects of type `Plane`)

4. `Flying.plane = new Plane();`

Exercise 7.s)

Which modifiers are implicitly added to all the methods declared in an interface (it is possible to choose more than one answer)?

1. `public`

2. `protected`

3. `private`

4. `static`

5. `default`

6. `abstract`

7. `final`

Exercise 7.t)

Given the following hierarchy:

```
interface A {
    void method();
}

interface B extends A {}

abstract class C implements B {}

public final class D extends C {
    public void method(){}
}
```

Which of the following statements are false (it is possible to choose more than one statement):

1. Class `C` cannot be declared abstract because it implements an interface, so this code does not compile.

2. Interface `B` cannot extend another interface.

3. Class `C` implementing `B` also inherits the abstract method `method` of `A`.

4. Class `D` does not compile because it cannot be declared `final`.

5. Class `D` does not compile because it is declared `public`.

6. Class `D` does not compile because its method is declared `public`.

Exercise 7.u)

Which of the following statements are true (it is possible to choose more than one statement):

1. Static methods declared in an interface are not inherited in subinterfaces.

2. It is not possible to declare an abstract and final class because the two modifiers are not compatible with each other.

3. An abstract class must necessarily declare abstract methods.

4. Interfaces can also declare constructors.

5. Abstract classes can declare static methods.

Exercise 7.v)

Suppose we want to define an Athlete type. Suppose each athlete defines the methods run() and doTraining(). Suppose we also want to define more specific subclasses like SoccerPlayer, Runner and TennisPlayer. How would you define Athlete, as an interface or abstract class? Would you add other types?

Exercise 7.w)

Which of the following statements regarding the protected modifier are correct:

1. A protected class can only be instantiated within the same package in which it is defined, and in all of its subclasses even if defined in different packages.

2. If a class has a declared protected constructor, it can only be instantiated by the classes belonging to the same package.

3. A protected method is inherited from a subclass regardless of the package in which it is defined.

4. A protected variable can be used by all the classes that belong to the same package as the class that defines this variable.

5. A protected variable is inherited from a subclass regardless of the package in which it is defined.

6. A protected constructor is inherited from a subclass regardless of the package in which it is defined.

7. A subclass defined in a package different from the one to which its superclass belongs, can instantiate the latter and then use its protected members.

Starting from the solution of the Exercise 4.z:

1. Encapsulate the `Contact` and `PhoneBook` classes.

2. Implement the Singleton pattern for the `PhoneBook` class.

3. Update the `SearchContacts` class to make it work the same as before.

4. Create a `Special` class that extends `Contact`. Suppose that a special contact must also have a `ringtone`, and create the corresponding instance variable of type `String`.

5. In the PhoneBook class, also create an array of Special objects (call it specialContacts), similar to the contacts array that already exists. Also create a searchSpecialContactsByName() method, equivalent to the `searchContactsByName()` method already present in the class.

6. Create a `SearchSpecialContacts` class equivalent to the `SearchContacts` class of the Exercise 4.z, to test the correct functioning of the written code.

We do not recommend that you perform the following exercise using Notepad and the command line. With an IDE (or at least with EJE) you will save many minutes to complete the exercise.

Starting from the Exercise 7.y, perform the following refactoring techniques to use the reuse, inheritance and abstraction paradigms:

1. Make sure that the created classes do not contain duplicate code (reuse all code that can be reused).

2. Create a `Data` interface to be implemented for all classes representing data in the application, and the `Identifiable` interface that declares an abstract method `getID()` which returns an `int`.

195

3. Abstract the concept of *entity* by implementing the `Entity` abstract class, which implements the `Data` and `Identifiable` interfaces. An `Entity` object must have an automatically generated incremental identification number. This feature will help us to make our abstractions evolve in future exercises.

4. Use inheritance to relate this new class (and possibly the other interfaces) with the already existing classes of the project where applicable.

5. Create the package `phonebook.data`, `phonebook.ui` and `phonebook.business`, where we mean with `phonebook.data`, a package that must contain classes that represent the data of the application. The package `phonebook.business` instead, must contain the classes that contain the business logic of the application. Finally, the `phonebook.ui` package must contain the classes that represent the interface with which the user can interact.

6. Move the classes and the interfaces of the project in the packages that seem most appropriate accordingly. If you think it necessary, you can create other classes and packages and modify existing ones (you can also create superclasses, subclasses and utility classes).

7. Test that the `SearchSpecialsContacts` and `SearchContacts` classes still work correctly.

Exercise 7.z)

Let's continue the case study defined in section 5.5. We had taken a series of steps, which represented a possible process to follow in order to create a code of quality. In particular, we defined the following steps:

1. Use case analysis.

2. Definition of scenarios for each use case.

3. Define a high level deployment diagram representing the architecture.

4. Identify the key abstractions.

5. Verify the validity of key abstractions using iteration diagrams to validate scenario flows, using objects instantiated by key abstractions.

We have seen in the final exercises related to chapter 6, that other steps to be performed are:

6. Define the key abstraction on a class diagram.

7. Re-evaluate the class diagram by adding the essential details, and above all by thinking about responsibilities.

Doing the entire Logos program is too demanding (it would take weeks if not months of work), but we can focus on particular use cases, and carry on our process only on these use cases. The reader could then also iterate the steps made on each use case to complete the program piece by piece. Even if in the initial analysis of Logos the authentication process has not been identified, actually there must be! In fact, two actors were defined by the analysis of the use cases: the administrator (who had the task of configuring the system) and the clerk (who had operational tasks). It seems obvious that in order to be recognized by the system, a login mechanism is indispensable. So, we can say that we have discovered a new use case, which we define as "authentication". We then evolve the diagram of the use cases of Figure 5.4 in the diagram of Figure 7.z.1, where we introduce the new case of use found.

For now, let's focus on the authentication use case in the context of Logos. We have the advantage of having already worked on a program that manages authentication with a certain flow, let's see if we can evolve it.

So, keeping in mind that we have already created the User class, we want to define the Clerk and Administrator classes. Should they be created? And why?
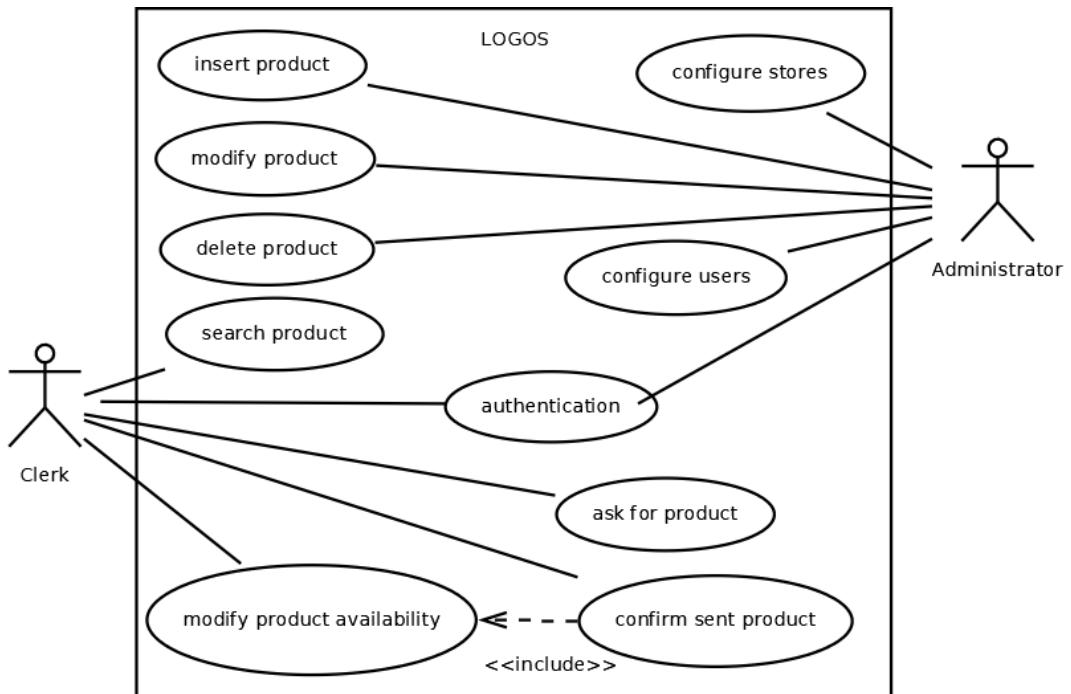
Figure 7.z.1 – Updated Logos Use case diagram.

Chapter 7
Exercise Solutions

Inheritance and Interfaces

1. **False**, the generalization process involves writing an extra class, and this does not always mean writing less code.

2. **False**.

3. **False**, even if the generalization process not always allow us to save code, it still has the advantage of making us manage the classes in a more natural way, favoring the abstraction of data. It also favor the implementation of polymorphism.

4. **True**.

5. **False**, multiple inheritance exists in the real world, and in Java exists just a soft version of it, because only the functional part of our entities can be inherited.

6. **False**, is an interface that declares a single abstract method.

7. **True**.

8. **False**, a team "is not a" player, nor a player "is a" team. If anything, a team "has" a player but this is not the relationship of inheritance. It is in fact the *association* relationship.

9. **True**, in fact both classes could extend a `Participant` class.

10. **False**, a `Father` is always a `Son`, or both could extend the `Person` class.

Solution 7.b)

```java
public class Employee extends Person { //inheritance
    private int id;

    public void setData(String name, int id) {
      setName(name); //reuse and inheritance
      setId(id); //reuse
    }

    public void setId(int id) {
        this.id = id; //encapsulation
    }

    public int getId() {
      return id; //encapsulation
    }

    public String getData() {
        //reuse, encapsulation and inheritance
        return getName() +", id: "+ getId();
    }
}
```

Solution 7.c) Abstract Classes and Interfaces, True or False:

1. **True**, the abstract and final modifiers cannot be used together because an abstract class should be extended, while a final class cannot be extended. For this reason, the compiler will not allow the creation of a class declared abstract and final.

2. **True**, the code block that defines the class is missing.

3. **True**, a final interface does not make sense.

4. **False**.

5. **True**.

6. **True**.

7. **True**.

8. **False**, an interface cannot declare constructors because it cannot be instantiated.

9. **True**.

10. **True**.

200

Solution 7.d)

Person could be a superclass of Teacher and Student, for all the rest there is no inheritance.

> **Inheritance is always tested with the "is a" relationship. So, it is very simple to verify that this relationship cannot be confirmed for all other class pairs.**

Solution 7.e)

Undoubtedly the Person class could be an abstract class, but even Teacher and Student could be declared abstract if we wanted to extend them with classes such as EngineeringStudent or MathematicsProfessor.

Solution 7.f)

Assuming that all object-oriented choices are subjective, the abstract implementation is probably the most correct choice for such an abstract concept:

```
/**
 * Abstracts the concept of a musical object.
 *
 * @author Claudio De Sio Cesari
 */
public interface Musical {
    /**
     * Performs the music of the current musical object.
     */
    void play();
}
```

Solution 7.g)

The code of the MusicalInstrument interface could be the following:

```
/**
 * Abstracts the concept of a musical instrument.
 *
 * @author Claudio De Sio Cesari
 */
public interface MusicalInstrument extends Musical {

}
```

The Ringtone interface code could be the following:

```
/**
 * Astrae the concept of musical ringtone.
 *
 * @author Claudio De Sio Cesari
 */
public interface Ringtone extends Musical {

}
```

The play() method for us is still abstract.

Solution 7.h)

While it might be plausible to consider a smartphone that uses a specific app a musical instrument, it is simply incorrect that it can be considered a ringtone, in fact the "is a" test fails:
Question: "Is a smartphone a musical instrument?"
Answer: Yes (as long as a specific app to play music is installed)
Question: "Is a smartphone a ringtone?"
Answer: No (if anything, it contains ringtones)

Solution 7.i) Interfaces after Java 8, True or False:

1. **True**.

2. **True**, the static modifier cannot be applied to classes and interfaces.

3. **True**, class is a keyword, and cannot be used as identifier.

4. **False**.

5. **False**.

6. **True**, in particular it is possible to rewrite a method with the same signature in a subinterface (but the same concept applies to classes), but technically it is not an override, because static methods are simply not inherited. In fact, any use of the @Override annotation in the subinterface to mark the static method will cause an error in compilation (see **SuperInterface.java** and **SubInterface.java**).

7. **True**, the rule "class always win" that we have seen in section 7.4.5.5, is true even if the class is abstract (see list **A.java**, **B.java**, **C.java**, **D.java**, **TestABCD.java**).

8. **False**, it could be misleading that the abstract class G also inheriting the abstract method from interface E makes the rule "class always win" studied in section 7.4.5.5 applies. In-

stead, in this case, rule "the most specific implementation wins" applies. In fact, class G inherits an E interface method, which is less specific than the one redefined in interface F (see list **E.java**, **F.java**, **G.java**, **H.java**, **TestEFGH.java**).

9. **True**, although some doubts may arise, an interface can extend multiple interfaces (see. **MyInterface.java**).

10. **False**, an interface cannot extend a class in any case.

Solution 7.j)

The abstraction of the Album class that we have created, is very similar to the abstraction of the Book class:

```java
public class Album extends Item {
    private String ismn;
    private String title;
    private String artist;
    private String genre;
    private String price;

    public Album(String ismn, String title, String artist, String genre, String price) {
        setIsmn(ismn);
        setTitle(title);
        setArtist(artist);
        setGenre(genre);
        setPrice(price);
    }

    public String getIsmn() {
        return ismn;
    }

    public void setIsmn(String ismn) {
        this.ismn = ismn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getArtist() {
        return artist;
    }
}
```

```
    public void setArtist(String artist) {
        this.artist = artist;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }

    public String getPrice() {
        return price;
    }

    public void setPrice(String price) {
        this.price = price;
    }
}
```

Note that there are three fields that are clearly in common with the Book class. For the variables price and title there is no doubt, while for the genre variable someone could have one since the musical genres do not correspond to the literary ones. Actually, having managed the possible values of the literal genres within the GenreUtils utility class, the genre variable does not depend directly on the values it can assume. Moreover, note that the variables isbn and ismn, represent substantially the same concept: a number that uniquely identifies an object. But also the variables author in Book and artist in Album, are semantically very similar in the context of the sale of books and music albums. Although there are differences in the meaning of the two words, a buyer of a book usually identifies it also through its author. For example, the book "Shining" is by Stephen King. In the same way a buyer of a musical album could identify an album also specifying the relative artist, for example we could say that the album "The Wall" is by Pink Floyd. This is true because we are in the context of selling albums and books, which for us are simply "items". So, we decided to create the abstract superclass Item of the Book and Album classes in the following way:

```
public abstract class Item {
    private String id;
    private String title;
    private String name;
    private int price;
    private String genre;
```

204

```java
    public Item(String id, String title, String name, int price, String genre) {
        super();
        setId(id);
        setTitle(title);
        setName(name);
        setPrice(price);
        setGenre(genre);
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }
}
```

The Book and Album classes can therefore be simplified in the following manner:

```
public class Book extends Item {
    public Book(String isbn, String title, String author, int price,
            String genre) {
        super(isbn, title, author, price, genre);
    }
}
public class Album extends Item {
    public Album(String ismn, String title, String artist, int price,
            String genre) {
        super(ismn, title, artist, price, genre);
    }
}
```

Note that we have "normalized" the isbn fields of the Book class and ismn of the Album class, as an id field in the Item superclass. The same goes for the author field of the Book class and artist field of the Album class, "normalized" in a name field in the superclass Item.

Solution 7.k)

With a non-simple solution, we used generalization for the utility classes. In particular we have extended with the classes MusicalGenreUtils and LiteraryGenreUtils a new generic class for which we have reused the name GenreUtils. In the latter, we implemented the static methods isValidGenre() and printError() by adding the array of genres on which to base the required check as second parameter. This is necessary, because we do not know a priori if in this generic class we will use musical or literary genres, but at the same time we do not want to duplicate code in the subclasses:

```
public class GenreUtils {

    public static boolean isValidGenre(String genre, String[] validGenres) {
        boolean validGenre = false;
        for (String fixedGenre : validGenres) {
            if (fixedGenre.equals(genre)) {
                validGenre = true;
            }
        }
        return validGenre;
    }

    public static void printError(String genre, String[] validGenres) {
        System.out.println("Genre " + genre +
            " not valid! Please, use one of the following genres:");
        for (String fixedGenre : validGenres) {
            System.out.println(fixedGenre);
        }
    }
}
```

So, we defined the `LiteraryGenreUtils` class in the following way:

```
public class LiteraryGenreUtils extends GenreUtils {
    public static final String ROMANCE = "Romance";
    public static final String ESSAY = "Essay";
    public static final String THRILLER = "Thriller";
    public static final String HANDBOOK = "Handbook";
    public static final String SCIFI = "Scifi";
    public static final String[] genres = { ROMANCE, ESSAY, THRILLER, HANDBOOK,
                                            SCIFI };

    public static boolean isValidGenre(String genre) {
        return isValidGenre(genre, genres);
    }

    public static void printError(String genre) {
        printError(genre, genres);
    }
}
```

And the `MusicalGenreUtils` class like this:

```
public class MusicalGenreUtils extends GenreUtils {
    public static final String ROCK = "Rock";
    public static final String JAZZ = "Jazz";
    public static final String BLUES = "Blues";
    public static final String POP = "Pop";
    public static final String RAP = "Rap";
    public static final String[] genres = { ROCK, JAZZ, BLUES, POP, RAP };

    public static boolean isValidGenre(String genre) {
        return isValidGenre(genre, genres);
    }

    public static void printError(String genre) {
        printError(genre, genres);
    }
}
```

In these two classes we have ensured that the `isValidGenre()` and `printError()` methods invoke the methods defined in the superclass. We also defined information on genres.
In the `Book` class, we then redefined the `setGenre()` method as can be seen below (in bold):

```
public class Book extends Item {

    public Book(String isbn, String title, String author, int price,
            String genre) {
        super(isbn, title, author, price, genre);
    }
```

```
    public void setGenre(String genre) {
        if (LiteraryGenreUtils.isValidGenre(genre)) {
            super.setGenre(genre);
        } else {
            LiteraryGenreUtils.printError(genre);
        }
    }
}
```

Note that we were forced to invoke the setGenre() method of the Item superclass using the super reference, since we needed to call a method that had the same name as the method we were redefining.

The same goes for the Album class:

```
public class Album extends Item {
    public Album(String ismn, String title, String artist, int price,
            String genre) {
        super(ismn, title, artist, price, genre);
    }

    public void setGenre(String genre) {
        if (MusicalGenreUtils.isValidGenre(genre)) {
            super.setGenre(genre);
        } else {
            MusicalGenreUtils.printError(genre);
        }
    }
}
```

Finally, we implemented the ItemsTest class in the following way:

```
public class ItemsTest {
    public static void main(String[] args) {
        Book jfaVol1 = new Book("979-12-200-4915-3", "Java for Aliens Vol. 1",
            "Claudio De Sio Cesari", 25, LiteraryGenreUtils.HANDBOOK);
        Book jfaVol2 = new Book("979-12-200-4916-0", "Java for Aliens Vol. 2",
            "Claudio De Sio Cesari", 25, "NonExisting");
        System.out.println("Genre JFA Vol 1 = " + jfaVol1.getGenre());
        System.out.println("Genre JFA Vol 2 = " + jfaVol2.getGenre());
        Album lad = new Album("979-0-236-44-3", "Live after Death",
            "Iron Maiden", 25, MusicalGenreUtils.ROCK);
        Album mop = new Album("978-0-789-01-2", "Master of Puppets",
            "Metallica", 25, "NonExisting");
        System.out.println("Genre Live after Death = " + lad.getGenre());
        System.out.println("Genre Master of Puppets = " + mop.getGenre());
    }
}
```

And this is the final output:

208

```
Genre NonExisting not valid! Please, use one of the following genres:
Romance
Essay
Thriller
Handbook
Scifi
Genre JFA Vol 1 = Handbook
Genre JFA Vol 2 = null
Genre NonExisting not valid! Please, use one of the following genres:
Rock
Jazz
Blues
Pop
Rap
Genre Live after Death = Rock
Genre Master of Puppets = null
```

Solution 7.l)

The compiler will actually transform the class into something very similar to the following (the implicit compiler entries are in bold):

```java
import java.lang.*;

public class CompilerItsYourTurn extends Object {

    private int var;

    public CompilerItsYourTurn() {
    }

    public void setVar(int v) {
      this.var = v;
    }

    public int getVar() {
      return this.var;
    }
}
```

> We also consider that, by extending `Object`, this class also inherits all of its methods.

Solution 7.m)

The correct answer is the first one. No error to report.

Solution 7.n)

The correct answer is the first one. No error to report.

Solution 7.o)

The concept of *aggregation*, used several times in previous exercises, is a relationship that indicates containment, which, if anything, can be considered an alternative to extension. So, the only correct answer is the third one.

Solution 7.p)

The fourth and fifth options are both correct for the compiler. The fifth, however, has less sense from a logic point of view (is an animal a lion? Not necessarily!).

Solution 7.q)

Even if an interface can potentially define methods of different types, it cannot declare instance variables, but only public static constants.

Solution 7.r)

The correct statements are the numbers 1 and 3.

> The `equals()` method is inherited from the `Object` class.

Solution 7.s)

Only the first answer is correct. Being the first correct, obviously the second and the third cannot be correct. Static and default methods can be declared, but their modifiers are never added automatically. The doubt can come for the answer 6, because before the advent of Java 8 this answer would have been right. But now we can declare in the interfaces also default and static methods. Finally, the `final` modifier is implicitly added to the attributes of the interfaces (which are also implicitly declared static and public).

210

Solution 7.t)

The answers are all false except the third.

Solution 7.u)

The correct answers are 1, 2 and 5.

Solution 7.v)

We could follow the following reasoning. A tennis player trains and runs differently from a soccer palyer or a runner. In short, both the `TennisPlayer` class, the `Runner` class and the `SoccerPlayer` class will redefine the `doTraining()` and `run()` methods. In the `Athlete` class instead, these two methods should be defined as abstract, because, a priori, it would be difficult to define how an athlete trains or runs, it depends on the type of athlete! At this point one might think of defining `Athlete` as an interface since it uses only two abstract methods, and this is feasible. But with the evolution of this program, it is highly probable that we will define athletes' fields as name and surname. An interface, however, cannot declare variables, and therefore one might prefer to declare an athlete as an abstract class. Or we could create a solution where `TennisPlayer`, `SoccerPlayer` and `Runner` implement the `Athlete` interface (which defines the two abstract methods `doTraining()` and `run()`) and extend the abstract class `Person` (which defines the attributes `name`, `surname`, etc.). In short, it will be the context of the program in which we will settle that will lead us to implement the most correct solution.

Solution 7.w)

The correct statements are 2, 3, 4, 5. The number 1 is incorrect because the `protected` modifier cannot be used with classes (there are no protected classes!). The 6 is false because the constructor are not inherited (even if they are public). The number 7 is incorrect for the reasons explained in section 7.2.5.1.

Solution 7.x)

The encapsulated `Contact` class could be the following (in bold the changes):

```java
public class Contact {
    protected static final String UNKNOWN = "unknown";
    private String name;
    private String phoneNumber;
    private String address;

    public Contact(String name, String phoneNumber) {
```

```
        this.setName(name);
        this.setPhoneNumber(phoneNumber);
        this.setAddress(UNKNOWN);
    }

    public Contact(String name, String phoneNumber, String address) {
        this.setName(name);
        this.setPhoneNumber(phoneNumber);
        this.setAddress(address);
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void printDetails() {
        System.out.println(name);
        System.out.println(address);
        System.out.println(phoneNumber);
        System.out.println();
    }
}
```

Note that we have introduced a private static constant UNKNOWN to represent an unknown address when using the first declared constructor (in the solutions of the Exercise 4.z, the address value was not really set, and therefore it was null).

The Singleton and encapsulated class PhoneBook could be implemented as follows (updates in bold):

```java
public class PhoneBook {
    private static PhoneBook instance;
    public Contact[] contacts;

    private PhoneBook () {
        contacts = new Contact[] {
            new Contact("Claudio De Sio Cesari", "13, Java Street",
              "131313131313"),
            new Contact("Stevie Wonder", "10, Music Avenue", "1010101010"),
            new Contact("Gennaro Capuozzo", "1, Four Days of Naples Square",
              "1111111111")
        };
    }

    public static PhoneBook getInstance() {
        if (instance == null) {
            instance = new PhoneBook();
        }
        return instance;
    }

    public Contact[] searchContactsByName(String name) {
        Contact []contactsFound = new Contact[contacts.length];
        for (int i = 0, j = 0;  i < contactsFound.length; i++) {
            if (contacts[i].getName().toUpperCase().
              contains(name.toUpperCase())) {
                contactsFound[j] = contacts[i];
                j++;
            }
        }
        return contactsFound;
    }

    public Contact[] getContacts() {
        return contacts;
    }
}
```

Note that we have avoided creating the setter method for the `contacts` variable.
We then replaced within the class `SearchContacts`, only the line:

```java
var phoneBook = new PhoneBook();
```

with the following:

```java
var phoneBook = PhoneBook.getInstance();
```

And everything continued to work as it worked before.
The `Special` class could be implemented like this:

```
public class Special extends Contact {

    private String ringtone;

    public Special(String name, String phoneNumber, String address,
      String ringtone) {
        super(name, phoneNumber, address);
        setRingtone(ringtone);
    }

    public String getRingtone() {
        return ringtone;
    }

    public void setRingtone(String ringtone) {
        this.ringtone = ringtone;
    }

    public void printDetails() {
        System.out.println(getName());
        System.out.println(getAddress());
        System.out.println(getPhoneNumber());
        System.out.println(getRingtone());
        System.out.println();
    }
}
```

Note that to instantiate an object of type Special, we can only use a constructor that takes as an input four parameters: name, phoneNumber, address and ringtone (since in our abstraction, it is the ringtone that should distinguish a special contact from a normal contact). This constructor calls the constructor of the Contact superclass via the super keyword. Also note that the printDetails() method has been redefined by adding the printout of the ringtone variable as well.

We then updated the PhoneBook class as required (updates in bold):

```
public class PhoneBook {
    private static PhoneBook instance;
    public Contact[] contacts;
    public Special[] specialContacts;

    private PhoneBook () {
        contacts = new Contact[] {
            new Contact("Claudio De Sio Cesari", "13, Java Street",
              "131313131313"),
            new Contact("Stevie Wonder", "10, Music Avenue", "1010101010"),
            new Contact("Gennaro Capuozzo", "1, Four Days of Naples Square",
              "1111111111")
        };
```

```java
        specialContacts = new Special[] {
            new Special("Mario Ruoppolo", "Neruda Street, 3", "333333",
              "The Postman"),
            new Special("Vincenzo Malinconico", "Courts Street, 8", "888888",
              "Tuca Tuca"),
            new Special("Logan Howlett", "Canada Square, 6", "66666", "Hurt")
        };
    }

    public static PhoneBook getInstance() {
        if (instance == null) {
            instance = new PhoneBook();
        }
        return instance;
    }

    public Contact[] searchContactsByName(String name) {
        Contact []contactsFound = new Contact[contacts.length];
        for (int i = 0, j = 0;  i < contactsFound.length; i++) {
            if (contacts[i].getName().toUpperCase().
              contains(name.toUpperCase())) {
                contactsFound[j] = contacts[i];
                j++;
            }
        }
        return contactsFound;
    }

    public Special[] searchSpecialContactsByName(String name) {
        Special []specialContactsFound = new Special[specialContacts.length];
        for (int i = 0, j = 0;  i < specialContactsFound.length; i++) {
            if (specialContacts[i].getName().toUpperCase().
              contains(name.toUpperCase())) {
                specialContactsFound[j] = specialContacts[i];
                j++;
            }
        }
        return specialContactsFound;
    }

    public Contact[] getContacts() {
        return contacts;
    }

    public Special[] getSpecialContacts() {
        return specialContacts;
    }
}
```

And finally we created the following class `SearchSpecialContacts`:

```java
import java.util.Scanner;

public class SearchSpecialContacts {
    public static void main(String args[]) {
        System.out.println("Search Special Contacts");
        System.out.println();
        var phoneBook = PhoneBook.getInstance();
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Special[] specialContactsFound =
          phoneBook.searchSpecialContactsByName(input);
        System.out.println("Special Contacts found with name containing \"" +
          input + "\"");
        for (Special special : specialContactsFound) {
            if (special != null) {
                special.printDetails();
            }
        }
    }
}
```

Solution 7.y)

In the `phonebook.data` package, we decided to put the three new requested abstractions `Data`:

```java
package phonebook.data;

public interface Data {
}
```

`Identifiable`:

```java
package phonebook.data;

public interface Identifiable {
    int getId();
}
```

and `Entity`:

```java
package phonebook.data;
import phonebook.util.Counter;

public abstract class Entity implements Data, Identifiable {
    private int id;
```

```
    public Entity () {
        setId(Counter.getSerialNumber());
    }

    private void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

Notice how this last class uses a utility class `Counter` (belonging to a `phonebook.util`, an ad hoc created package) that contains a static method that updates a serial number every time an entity is instantiated (i.e. a subclass, since `Entity` is an abstract class). The `Counter` class follows:

```
package phonebook.util;

public class Counter {
    private static int objectCounter;

    public static int getSerialNumber() {
        return objectCounter += 1;
    }
}
```

> **As already mentioned, the serial number will be useful in future exercises.**

We decided to include the `Contact` and `PhoneBook` classes in the `phonebook.data` package. With regard to the latter, we deemed it necessary to modify it as follows:

```
package phonebook.data;

public class PhoneBook implements Data {
    private static PhoneBook instance;
    public Contact[] contacts;
    public Special[] specialContacts;

    private PhoneBook () {
        contacts = new Contact[] {
            new Contact("Claudio De Sio Cesari", "13, Java Street",
              "131313131313"),
            new Contact("Stevie Wonder", "10, Music Avenue", "1010101010"),
            new Contact("Gennaro Capuozzo", "1, Four Days of Naples Square",
              "1111111111")
        };
```

```
        specialContacts = new Special[] {
            new Special("Mario Ruoppolo", "Neruda Street, 3", "333333",
              "The Postman"),
            new Special("Vincenzo Malinconico", "Courts Street, 8", "888888",
              "Tuca Tuca"),
            new Special("Logan Howlett", "Canada Square, 6", "66666", "Hurt")
        };
    }

    public static PhoneBook getInstance() {
        if (instance == null) {
            instance = new PhoneBook();
        }
        return instance;
    }

    public Contact[] getContacts() {
        return contacts;
    }

    public Special[] getSpecialContacts() {
        return specialContacts;
    }
}
```

Because searchContactsByName() and searchSpecialContactsByName() are business methods and not methods that the PhoneBook class should declare, we have removed them.

In fact, in our phone book abstraction, we refer to the concept of a phone book containing contacts, not a phone book that can perform searches. As for the old and now no longer used paper phone book, we mean an object similar to a notebook in which the contacts are written. A phone book therefore represents a data container and not an object that performs business actions, in fact we have made it implement the Data interface. To search for a certain contact in a paper phone book, a user must browse the pages of the phone book. So, we have decided for now to move the searchContactsByName() and searchSpecialContactsByName() methods in a new class called User, which will perform searches in the address book:

```
package phonebook.business;
import phonebook.data.*;

public class User {
    public Contact[] searchContactsByName(String name) {
        Contact[]contacts = PhoneBook.getInstance().getContacts();
        Contact []contactsFound = new Contact[contacts.length];
        for (int i = 0, j = 0;  i < contactsFound.length; i++) {
            if (contacts[i].getName().toUpperCase().
              contains(name.toUpperCase())) {
                contactsFound[j] = contacts[i];
                j++;
            }
        }
```

```
            return contactsFound;
        }

    public Special[] searchSpecialContactsByName(String name) {
        Special[]specialContacts = PhoneBook.getInstance().getSpecialContacts();
        Special []specialContactsFound = new Special[specialContacts.length];
        for (int i = 0, j = 0;  i < specialContactsFound.length; i++) {
            if (specialContacts[i].getName().toUpperCase().
              contains(name.toUpperCase())) {
                specialContactsFound[j] = specialContacts[i];
                j++;
            }
        }
        return specialContactsFound;
    }
}
```

Note that this class belongs to the phonebook.business package.

Summarizing, we abstract a phonebook as data, we make them implement the Data interface and move it in the phonebook.data package, while the User class represents a business object and belongs to the phonebook.business package.

Finally, we have modified the SearchContacts and SearchSpecialContacts classes in the following way (updates in bold):

```
package phonebook.ui;

import phonebook.data.*;
import phonebook.business.User;
import java.util.Scanner;

public class SearchContacts {
    public static void main(String args[]) {
      System.out.println("Search Contacts");
        System.out.println();
        var user = new User();
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Contact[] foundContacts = user.searchContactsByName(input);
        System.out.println("Contacts found with name containing \"" +
          input + "\"");
        for (Contact contact : foundContacts) {
            if (contact != null) {
                contact.printDetails();
            }
        }
    }
}
```

```
package phonebook.ui;

import phonebook.data.*;
import phonebook.business.User;
import java.util.Scanner;

public class SearchSpecialContacts {
    public static void main(String args[]) {
        System.out.println("Search Special Contacts");
        System.out.println();
        var user = new User();
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Special[] specialContactsFound =
          user.searchSpecialContactsByName(input);
        System.out.println("Special Contacts found with name containing \""
          + input +"\"");
        for (Special special : specialContactsFound) {
            if (special != null) {
                special.printDetails();
            }
        }
    }
}
```

Solution 7.z)

A simple and almost automatic solution, would consist in extending the User class directly, with the two sub-classes Administrator and Clerk:

```
package com.claudiodesio.authentication;

public class Administrator extends User {
    public Administrator (String name, String username, String password) {
        super(name, username, password);
    }
}
```

and:

```
package com.claudiodesio.authentication;

public class Clerk extends User {
    public Clerk(String name, String username, String password) {
        super(name, username, password);
    }
}
```

Except that we still don't have enough information to insert specific fields and methods for these two classes (in fact they are empty). So for now we decide not to implement them.

Chapter 8
Exercises

Polimorphism

In this chapter we will simulate the construction of an IDE, in a very simplified way. Also in this case, it will be an incremental exercise so you have to complete each exercise (and read the solution) in order to move on to the next one.

We will create classes and interfaces step by step. In particular we will create the abstractions of `IDE`, `Editor`, `SourceFile`, `File`, `FileType` and so on. The exercise is guided, so the reader is relieved of the responsibility to decide which classes should compose our application. For this reason, there are exercises on programming rather than analysis and design.

With the next exercises we will only have to apply the definitions we have learned so far, almost no algorithm is required. The ultimate goal is to focus on the Object Orientation and not on the algorithms. In addition, other types of exercises are also presented, such as those that support preparation for Oracle certifications.

Exercise 8.a) Polymorphism for Methods, True or False:

1. Method overloading implies writing another method with the same name and a different return type.

2. Method overloading implies writing another method with a different name and the same list of parameters.

3. The signature of a method consists of the identifier - parameter list pair.

4. To take advantage of method overriding, inheritance must be in place.

5. To take advantage of method overloading, inheritance must be in place.

6. Suppose that the class B, which extends the class A, inherits the following method:

```
public int m(int a, String b) { ... }
```

If in the B class, we write the following method:

```
public int m(int c, String b) { ... }
```

we are doing method overloading and not overriding.

7. If in the B class, we write the following method:

```
public int m(String a, String b) { ... }
```

we are doing method overloading and not overriding.

8. If in the B class, we write the following method:

```
public void m(int a, String b) { ... }
```

we will get a compilation error.

9. If in the B class, we write the following method:

```
protected int m(int a, String b) { ... }
```

we will get a compilation error.

10. If in the B class, we write the following method:

```
public int m(String a, int c) { ... }
```

we are doing a method overriding.

Exercise 8.b) Polymorphism for Data, True or False:

1. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Vehicle v [] = {new Car(), new Plane(), new Plane()};
```

2. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Object o [] = {new Car(), new Plane(), "ciao"};
```

3. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Plane a [] = {new Vehicle(), new Plane(), new Car()};
```

4. Considering the classes introduced in this chapter, and if the method of the `Traveler` class was the following:

```
public void travel(Object o) {
    o.accelerate();
}
```

we could pass as parameter to it an object of type Vehicle without having compilation errors. For example:

```
claudio.travel(new Vehicle());
```

5. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
ThreeDimensionalPoint ogg = new Point();
```

6. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
ThreeDimensionalPoint ogg = (ThreeDimensionalPoint)new Point();
```

7. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Point ogg = new ThreeDimensionalPoint();
```

8. Considering the classes introduced in this chapter, and if the `Piper` class extends the `Plane` class, the following snippet will not produce compilation errors:

```
Vehicle a = new Piper();
```

9. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
String string = fiat500.toString();
```

10. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
public void payEmployee(Employee emp) {
    if (emp instanceof Employee) {
        emp.setSalary(1000);
    } else if (emp instanceof Programmer) {
        ...
    }
```

223

Exercise 8.c)

To start creating a simple IDE, create a `FileType` interface that defines static constants that represent the types of source files that our IDE should manage. Certainly, one of these constants must be called `JAVA`. Choose all the others as you like. Also choose the type of constants as you like.

Exercise 8.d)

Create a `File` class that abstracts the concept of a generic file and defines a name and a type.

Exercise 8.e)

Create a `SourceFile` class that abstracts the concept of source file (extending the `File` class) which also defines a string type content.

Exercise 8.f)

Add an `addText()` method, that adds a text string to the end of the contents of the source file.

Exercise 8.g)

Add an overloaded `addText()` method, that adds a `text` string to a specified point in the contents of the source file (see the `String` class documentation).

Exercise 8.h)

Create a `SourceFileTest` class that tests the correct operation of the `SourceFile` class.

Exercise 8.i)

Create an `Editor` interface that abstracts the concept of text editor. You need to define methods to open, close, save and edit a file.

Exercise 8.j)

Create an `IDE` interface that abstracts the concept of IDE. Please note that an IDE is also an editor. You need to define methods to compile and execute a file.

Exercise 8.k)

Create a simple `JavaIDE` implementation of the `IDE` interface. Add an implementation for the `edit()` method (and as you wish, you can re-implement all the methods that you find useful).

224

Exercise 8.l)

Starting from the solutions of the Exercise 7.k, in which we created classes to simulate a bookshop, perform the following steps:

1. Create the `bookshop.data`, `bookshop.business`, `bookshop.util` and `bookshop.interface` packages, and move the existing classes within the packages that are deemed most appropriate.

2. For the `Book` and `Album` classes, create the `toString()` method, which returns a one line description of the item. The `toString()` method should print this information: `article_type: (isbn) - "title" by author, genre, price`. For example: `Book: (2345) "The Pillars of the Earth" by Ken Follet, Romance, 18€`,

3. Modify the `ItemTest` class, so that it also test the `toString()` methods added.

Exercise 8.m)

Starting from the solution of the previous exercise, create a `ShoppingCart` class, which represents a shopping cart. This abstraction must declare the methods: `add()` to add an item to the shopping cart, `calculatePrice()` which returns the total price of the items in the shopping cart, `toString()` which prints the description of the items in the shopping cart. The shopping cart can contain a maximum of four items (books or albums). Make sure that the `add()` method prints an error when trying to add the eleventh article, and that the element is consequently not added.

Create the `ShoppingCartTest` class, to test the correct functioning of the `ShoppingCart` class, invoking all the defined methods, and try adding five items to verify the correct functioning of the check implemented in the `add()` method.

Exercise 8.n)

Create a `IDETest` test class that performs file operations using IDE.

> The exercise could continue extending these classes further, feel free to do other programming iterations after completing this one. You will have to perform three steps: provide yourself specifications, understand how to implement them and implement them.

225

Exercise 8.o) Varargs, True or False:

1. The varargs allow you to use the methods as if they were overloads.

2. The following declaration can be compiled without errors:

```java
public void myMethod(String... s, Date d) {
   ...
}
```

3. The following declaration can be compiled without errors:

```java
public void myMethod(String... s, Date d...) {
   ...
}
```

4. Considering the following method:

```java
public void myMethod(Object... o) {
   ...
}
```

the following invocation is correct:

```java
oggetto.myMethod();
```

5. The following declaration can be compiled without errors:

```java
public void myMethod(Object o, Object os...) {
   ...
}
```

6. Considering the following method:

```java
public void myMethod(int i, int... is) {
   ...
}
```

the following invocation is correct:

```java
object.myMethod(new Integer(1));
```

7. The rules of overriding change with the introduction of varargs.

8. The `printf()` method of the `java.io.PrintStream` class is based on the `format()` method of the `java.util.Formatter` class.

9. The `format()` method of the `java.util.Formatter` class has no overload because it is defined with a varargs.

10. In case you pass an array as varargs to the `printf()` method of `java.io.PrintStream`, this will be treated not as a single object but as if each of its elements had been passed to one by one.

Exercise 8.p)

Given the following hierarchy:

```java
interface A {
    void method();
}

interface B implements A {
    static void staticMethod() {}
}

final class C implements B {}

public abstract class D implements A {
    @Override
    void method() {}
}
```

Choose all the true statements:

1. Interface `C` does not inherit the `staticMethod()` method.

2. Interface `B` cannot implement another interface.

3. The class `C` implementing `B` also inherits the abstract `method()` method of the `A` interface, and since it is not declared `abstract` it cannot be compiled.

4. Class `D` cannot be compiled because it cannot be declared `abstract`. In fact, it does not declare any abstract method.

5. Class `D` does not compile because the method it declares is not declared `public`.

6. Class `D` compiles only because the method is annotated with `Override`.

Exercise 8.q)

Which of the following statements is correct (choose all that apply):

1. An interface extends the class `Object`.

2. A method that takes as a parameter a reference of type `Object`, can take as input any object of any type, even of an interface type.

3. A method that takes as a parameter a reference of type `Object`, can take as input any object of any type, even an array.

4. A method that takes as a parameter a reference of type `Object`, can take as an input any object of any type, even a heterogeneous collection.

5. All casts of objects are evaluated at compile time

Exercise 8.r)

Consider the following classes:

```java
public class PrintNumber {
    public void print(double number){
        System.out.print(number);
    }
}

public class PrintInteger extends PrintNumber{
    public void print(int number) {
        System.out.print(number);
    }

    public static void main(String args[]) {
        PrintNumber printNumber = new PrintInteger();
        printNumber.print(1);
    }
}
```

If we run the `PrintInteger` class, what will be the output?

1. 1.2

2. 1

3. 1.0

4. 11.2

Exercise 8.s)

Consider the following hierarchy:

```java
public interfac7e Satellite {
    void orbit();
}
```

```
public class Moon implements Satellite{
    @Override
    public void orbit(){
        System.out.println("Moon is orbiting");
    }
}
public class ArtificialSatellite implements Satellite {
    @Override
    public void orbit() {
        System.out.println("Artificiale satellite is orbiting");
    }
}
```

And the following class of tests:

```
public class SatellitesTest {
    public static void main(String args[]) {
        test(new Moon(), new ArtificialSatellite());
        Satellite[] satellites = {
            new Moon(), new ArtificialSatellite()
        } ;
        test(satellites);
        test();
  //   test(new Object());
    }

    public static void test(Satellite... satellites) {
        for (Satellite satellite : satellites) {
            satellite.orbit();
        }
    }
}
```

Choose all the correct statements:

1. The application compiles and runs without errors.

2. The application does not compile because of the instruction `test(satellites);` .

3. The application does not compile for the instruction `test();`.

4. The application does not compile for the instruction `test(new Object());`.

5. The application compiles but it, due to an exception, crashes at runtime.

Exercise 8.t)

Define the overload and the override concepts. And give an example of a subclass, which implements both concepts.

229

Exercise 8.u)

Bearing in mind that `Number` is superclass of the `Integer` class, let us consider the following hierarchy:

```
public abstract class SumNumber {
    public abstract Number sum(Number n1, Number n2);
}
public class SumInteger extends SumNumber{
    @Override
    public Integer sum(Number n1, Number n2) {
        return (Integer)n1 + (Integer)n2;
    }
}
```

Choose all the correct statements:

1. The `SumInteger` class compiles without errors.

2. The `SumInteger` class does not compile because the override is not correct: the return types do not coincide.

3. The `SumInteger` class does not compile because it is not possible to use the + operator except with primitive type numbers.

4. The `SumInteger` class could cause an exception at runtime.

Exercise 8.v)

Make the `sum()` method of the `SumInteger` class defined in exercise 8.u robust, so that the runtime works without exceptions.

Exercise 8.w)

Briefly define what a polymorphic parameter is, what heterogeneous collections are, and what a virtual call to a method is.

Exercise 8.x)

Starting from the solution of the Exercise 7.y, let's evolve our application that simulates a phone book:

1. Replace the `printDetails()` method with the override of the `toString()` method, in the `Contact` and `Special` classes. This change is consistent, because these classes represent application data, and therefore should not "print details". Make sure that the `toString()` method also returns the identifier (`id` variable).

230

2. In the `PhoneBook` class, add the elements of the `specialContacts` array to the `contacts` array, and delete the `specialContacts` variable. Modify the `getContacts()` and `getSpecialContacts()` methods to make that the first return all contacts (both special and ordinary contacts), while the latter returns only the special contacts.

3. If necessary, modify the `User` class to make it work consistently with the new implementation of the `PhoneBook` class.

4. Modify the `SearchContacts`, and `SearchSpecialContacts` classes, so that they work in line with the changes made.

Exercise 8.y)

1. Starting from the solution of the previous exercise:

2. Add in the `PhoneBook` class a method called `getOrdinaryContacts()` that returns all ordinary contacts, that is, the contacts that are not of the `Special` type.

3. Add a method called `searchOrdinaryContactsByName()` in the `User` class, which searches by name only contacts that are not special.

4. Create a class `SearchOrdinaryContacts`, homologous to the other search classes we have already created.

Exercise 8.z)

1. Starting from the solution of the previous exercise:

2. Until now, we have assigned the responsibility for creating `Contact` objects explicitly to the class containing the `main()` method. So, now is the time to create a `ContactFactory` class that has this responsibility, declaring one or more static methods `getContact()`. These methods will take as input the necessary fields to instantiate the objects.

3. In the `User` class, add a method called `add()` which adds a `Contact` object to the `contacts` array. Pay attention that in the `PhoneBook` class, the `contacts` array can contain a finite number of `Contact` objects. Add a number of free places in the array to be filled with new objects.

4. Test with the `SearchContacts` class the correct functioning of the `add()` method of the `PhoneBook` class. Create the objects to pass to the method using the `ContactFactory` class.

231

Chapter 8
Exercise Solutions

Polimorphism

1. **False**, overloading a method implies writing another method with the same name and a different list of parameters.

2. **False**, overloading a method implies writing another method with the same name and a different list of parameters.

3. **True**.

4. **True**.

5. **False**, overloading a method implies writing another method with the same name and a different list of parameters.

6. **False**, we are overriding. The only difference lies in the name of the identifier of a parameter, which is irrelevant in order to distinguish methods.

7. **True**, the parameter list of the two methods is different.

8. **True**, when overriding the return type cannot be different.

9. **True**, when overriding the rewritten method cannot be less accessible than the original method.

10. **False**, we will get an overload. In fact, the two parameter lists differ in positions.

Solution 8.b) Polymorphism for Data, True or False:

1. **True**.

2. **True**.

3. **False**, the Vehicle class is abstract and is not instantiable. Furthermore, it is not possible to insert in a heterogeneous collection of planes an object of type Vehicle, which is a Plane superclass.

4. **False**, the compilation would fail already since we tried to compile the travel() method. In fact, it is not possible to call the accelerate() method with a reference of type Object.

5. **False**, there is need for a casting, because the compiler does not know a priori the type to which the reference at runtime will point.

6. **True**.

7. **True**.

8. **True**, in fact, Vehicle is a superclass of the Piper class.

9. **True**, the toString() method, belongs to all classes because it is inherited from the superclass Object.

10. **True**, but all employees will be paid the same way.

Solution 8.c)

The code could be the following:

```
public interface FileType {
    int JAVA = 1;
    int C_SHARP = 2;
    int C_PLUS_PLUS = 3;
    int C = 4;
}
```

Note that it is not necessary to specify modifiers for constants, since they are implicitly declared public, static and final. Moreover, we have chosen the int type as the type of the constants, but any other type would have been fine, the important thing is that the constants have different values.

> This type of use of interfaces has been in disuse for years, precisely since enumerations were introduced in Java version 5 (see Chapter 11). However, we will still use this programming style since we have not yet addressed the topic of enumerations.

Solution 8.d)

The code of the File class could be the following:

```java
public abstract class File {

    private String name;

    private int type;

    public File(String name, int type) {
        this.name = name;
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }
}
```

Note that we have declared the class as abstract, since it is generic and created for the purpose of extension.

Solution 8.e)

The code of SourceFile class should be as the following:

```
public class SourceFile extends File {
    private String content;

    public SourceFile(String name, int type) {
        super(name, type);
    }

    public SourceFile(String name, int type, String content) {
        this(name, type);
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

Note that we have reused the superclass constructor using the super keyword, and we have also created a constructor that is equivalent to the superclass constructor.

Solution 8.f)

The code of the requested method could be:

```
public void addText(String text) {
    if (content == null) {
        content ="";
    }
    if (text != null) {
        content += text;
    }
}
```

A null string "added" to another string is represented with the string "null", that's why we implemented the first nullity check in the method.

Solution 8.g)

The code of the requested method could be:

```
public void addText(String text, int position) {
    final int length = content.length();
    if (content != null && text != null && position > 0
        && position < length) {
```

236

```
            content = content.substring(0, position) + text +
            content.substring(position);
        }
    }
```

We have chosen, for simplicity, not to add text if the specified position is not correct. However, an else clause that prints an error message could be a better solution. Actually, this method can be improved a lot... try it!

> **In Chapter 9 we will see how to handle exceptions in Java.**

Solution 8.h)

The SourceFileTest source code could be the following:

```
public class SourceFileTest {
    public static void main(String args[]) {
        SourceFile sourceFile = new SourceFile("Test.java",
            FileType.JAVA, "public class MyClass {\n\r");
        System.out.println(sourceFile.getContent());
        // Test addText (String) correct
        sourceFile.addText("}");
        System.out.println(sourceFile.getContent());
        // Test addText (String,int) correct
        sourceFile.addText("//Test adding text\n\r", 23);
        System.out.println(sourceFile.getContent());
        // Test addText (String,int) incorrect
        sourceFile.addText("//Test adding text\n\r", -1);
        System.out.println(sourceFile.getContent());
        // Test addText (String,int) incorrect
        sourceFile.addText("//Test adding text\n\r", 100);
        System.out.println(sourceFile.getContent());
    }
}
```

The output will be the following:

```
public class MyClass {

public class MyClass {
}
public class MyClass {
//Test adding text
}
public class MyClass {
//Test adding text
}
```

```
public class MyClass {
//Test adding text
}
```

You could write many more test cases, and you should also make sure that each test case doesn't depend on the previous one, but for now it's okay.

> **To perform test cases with greater convenience, the use of a tool like JUnit is recommended. You can find a brief description of JUnit in the sections 10.4.1 and 10.4.2.**

Solution 8.i)

The code of the requested interface could be:

```
public interface Editor {
    default void save(SourceFile file) {
        System.out.println("File: " + file.getName() + " saved!");
    }
    default void open(SourceFile file) {
        System.out.println("File: " + file.getName() + " open!");
    }
    default void close(SourceFile file) {
        System.out.println("File: " + file.getName() + " closed!");
    }
    default void update(SourceFile file, String testo) {
        System.out.println("File: " + file.getName() + " updated!");
    }
}
```

We have created default methods that simulate the real execution just printing a simple sentence.

Solution 8.j)

The code of the requested interface could be:

```
public interface IDE extends Editor {
    default void compile(SourceFile file) {
        System.out.println("File: " + file.getName() + " compiled!");
    }

    default void execute(SourceFile file) {
        System.out.println("File: " + file.getName() + " executed!");
    }
}
```

238

Also in this case, we have created default methods that simulate the real execution just printing a simple sentence.

Solution 8.k)

The code of the requested class could be:

```
public class JavaIDE implements IDE {
    @Override
    public void update(SourceFile file, String text) {
        IDE.super.update(file, text);
        file.addText(text);
        System.out.println("Content updated:\n" + file.getContent());
    }
}
```

Solution 8.l)

The code of the requested class could be:

```
public class IDETest {
    public static void main(String args[]) {
        IDE ide = new JavaIDE();
        SourceFile sourceFile = new SourceFile("Test.java",
          FileType.JAVA, "public class MyClass {\n\r");
        ide.update(sourceFile, "}");
    }
}
```

The output will be:

```
File: Test.java updated!
Content updated:
public class MyClass {

}
```

Solution 8.m)

We moved the `Item`, `Album` and `Book` classes in the `bookshop.data` package, the `GenreUtils`, `LiteraryGenreUtils` and `MusicalGenreUtils` classes within the `bookshop.util` package, the `ItemsTest` class within the `bookshop.test` package, finally we leaved the `bookshop.business` package empty, since for now we don't yet have classes that encapsulate business logic.
To create the required `toString()` methods in the `Book` and `Album` classes, let's also take advantage of their abstract superclass `Item`, which we rewrite below (changes in bold):

239

```
package bookstore.data;
public abstract class Item {
    private static final char CURRENCY = '$';
    private String id;
    private String title;
    private String name;
    private int price;
    private String genre;

    public Item(String id, String title, String name, int price, String genre) {
        super();
        setId(id);
        setTitle(title);
        setName(name);
        setPrice(price);
        setGenre(genre);
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }
```

```
    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }

    @Override
    public String toString() {
        return ": (" + getId() + ") " + getTitle() + " di " + getName() +
            ", " + getGenre() + ", " + getPrice() + " " + CURRENCY;
    }
}
```

Note that we have also introduced a private and static CURRENCY constant to represent the price currency.

We can then rewrite the toString() method in the Book subclass as follows (changes in bold):

```
package bookstore.data;

import bookstore.util.*;

public class Book extends Item {

    public Book(String isbn, String title, String author, int price, String genre) {
        super(isbn, title, author, price, genre);
    }

    public void setGenre(String genre) {
        if (LiteraryGenreUtils.isValidGenre(genre)) {
            super.setGenre(genre);
        } else {
            LiteraryGenreUtils.printError(genre);
        }
    }

    @Override
    public String toString() {
        return "Book" + super.toString();
    }
}
```

And we can then rewrite the toString() method in the Album subclass in the following way (changes in bold):

```
package bookstore.data;

import bookstore.util.*;
```

241

```
public class Album extends Item {
    public Album(String ismn, String title, String artist, int price, String genre) {
        super(ismn, title, artist, price, genre);
    }

    public void setGenre(String genre) {
        if (MusicalGenreUtils.isValidGenre(genre)) {
            super.setGenre(genre);
        } else {
            MusicalGenreUtils.printError(genre);
        }
    }

    @Override
    public String toString() {
        return "Album" + super.toString();
    }
}
```

Finally let's modify the `ItemsTest` class in the following way (changes in bold):

```
package bookstore.test;

import bookstore.data.*;
import bookstore.util.*;

public class ItemsTest {
    public static void main(String[] args) {
        Book jfaVol1 = new Book("979-12-200-4915-3",
          "Java for Aliens Vol. 1","Claudio De Sio Cesari", 25,
            LiteraryGenreUtils.HANDBOOK);
        Book jfaVol2 = new Book("979-12-200-4916-0",
          "Java for Aliens Vol. 2","Claudio De Sio Cesari", 25,"NonExisting");
//        System.out.println("Genre JFA Vol 1 = " + jfaVol1.getGenre());
//        System.out.println("Genre JFA Vol 2 = " + jfaVol2.getGenre());
        System.out.println(jfaVol1);
        System.out.println(jfaVol2);
        Album lad = new Album("979-0-236-44-3","Live after Death","Iron Maiden",
          25, MusicalGenreUtils.ROCK);
        Album mop = new Album("978-0-789-01-2","Master of Puppets","Metallica",
          25,"NonExisting");
//        System.out.println("Genre Live after Death = "+ lad.getGenre());
//        System.out.println("Genre Master of Puppets = "+ mop.getGenre());
        System.out.println(lad);
        System.out.println(mop);
    }
}
```

Obviously the `toString()` method is automatically called by the `println()` method. The out-

242

put of the previous class will be the following:

```
Genre NonExisting not valid! Please, use one of the following genres:
Romance
Essay
Thriller
Handbook
Scifi
Book: (979-12-200-4915-3) Java for Aliens Vol. 1 di Claudio De Sio Cesari, Handbook, 25 $
Book: (979-12-200-4916-0) Java for Aliens Vol. 2 di Claudio De Sio Cesari, null, 25 $
Genre NonExisting not valid! Please, use one of the following genres:
Rock
Jazz
Blues
Pop
Rap
Album: (979-0-236-44-3) Live after Death di Iron Maiden, Rock, 25 $
Album: (978-0-789-01-2) Master of Puppets di Metallica, null, 25 $
```

Solution 8.n)

We could implement the ShoppingCart class in the following way:

```java
package bookstore.data;

import java.util.ArrayList;
import java.util.List;

public class ShoppingCart {
    private static final int ITEMS_MAX_NUMBER = 4;
    private Item[] items;

    public ShoppingCart() {
        items = new Item[ITEMS_MAX_NUMBER];
    }

    public void add(Item item) {
        for (int i = 0; i < items.length; i++) {
            if (items[i]==null) {
                items[i] = item;
                return;
            }
        }
        System.out.println("Cannot add Item: shopping cart full!");
    }

    public boolean isEmpty() {
        return items[0] == null;
    }
```

```
    public int calculatePrice() {
        int shoppingCartPrice = 0;
        for (Item item : items) {
            shoppingCartPrice += item.getPrice();
        }
        return shoppingCartPrice;
    }

    public String toString() {
        String shoppingCartDescription =
          "The shopping cart contains the following items:\n";
        for (Item item : items) {
            shoppingCartDescription += item + "\n";
        }
        return shoppingCartDescription;
    }
}
```

While we could implement the ShoppingCartTest class (within the bookshop.test package) in the following way:

```
package bookstore.test;
import bookstore.data.*;
import bookstore.util.*;

public class ShoppingCartTest {
    public static void main(String[] args) {
        Book jfaVol1 = new Book("979-12-200-4915-3", "Java for Aliens Vol. 1",
          "Claudio De Sio Cesari", 25, LiteraryGenreUtils.HANDBOOK);
        Book jfaVol2 = new Book("979-12-200-4916-0","Java for Aliens Vol. 2",
          "Claudio De Sio Cesari", 25, LiteraryGenreUtils.HANDBOOK);
        Album lad = new Album("979-0-236-44-3","Live after Death","Iron Maiden",
          25, MusicalGenreUtils.ROCK);
        Album mop = new Album("978-0-789-01-2","Master of Puppets","Metallica",
          25, MusicalGenreUtils.ROCK);
        Album tt = new Album("978-0-789-01-9","Tokyo Tapes","Scorpions", 22,
          MusicalGenreUtils.ROCK);
        ShoppingCart shoppingCart = new ShoppingCart();
        System.out.println("ShoppingCart empty = " + shoppingCart.isEmpty());
        System.out.println("Adding the book " + jfaVol1);
        shoppingCart.add(jfaVol1);
        System.out.println("Adding the book " + jfaVol2);
        shoppingCart.add(jfaVol2);
        System.out.println("Adding the album " + lad);
        shoppingCart.add(lad);
        System.out.println("Adding the album " + mop);
        shoppingCart.add(mop);
        System.out.println("Adding the album " + tt);
        shoppingCart.add(tt);
```

```
            System.out.println("ShoppingCart Empty = " + shoppingCart.isEmpty());
            System.out.println(shoppingCart);
        }
    }
```

The output of the `ShoppingCartTest` class follows:

```
ShoppingCart empty = true
Adding the book Book: (979-12-200-4915-3) Java for Aliens Vol. 1 di Claudio De Sio Cesari,
   Handbook, 25 $
Adding the book Book: (979-12-200-4916-0) Java for Aliens Vol. 2 di Claudio De Sio Cesari,
   Handbook, 25 $
Adding the album Album: (979-0-236-44-3) Live after Death di Iron Maiden, Rock, 25 $
Adding the album Album: (978-0-789-01-2) Master of Puppets di Metallica, Rock, 25 $
Adding the album Album: (978-0-789-01-9) Tokyo Tapes di Scorpions, Rock, 22 $
Cannot add Item: shopping cart full!
ShoppingCart Empty = false
The shopping cart contains the following items:
Book: (979-12-200-4915-3) Java for Aliens Vol. 1 di Claudio De Sio Cesari, Handbook, 25 $
Book: (979-12-200-4916-0) Java for Aliens Vol. 2 di Claudio De Sio Cesari, Handbook, 25 $
Album: (979-0-236-44-3) Live after Death di Iron Maiden, Rock, 25 $
Album: (978-0-789-01-2) Master of Puppets di Metallica, Rock, 25 $
```

Solution 8.o) Varargs, True or False:

1. **True**.

2. **False**.

3. **False**.

4. **True**.

5. **True**.

6. **True**.

7. **False**.

8. **True**.

9. **False**.

10. **True**.

Solution 8.p)

The true answers are the 1, the 2, the 3 and the 5. The number 5 is true because the inherited method is implicitly public, and redefining it without the `public` modifier, we are making it less

245

accessible than the inherited one. In fact, by compiling class D, we will get the following error:

```
error: metodo() in D cannot implement method() in A
  void method () {}
       ^
  attempting to assign weaker access privileges; was public
1 error
```

Solution 8.q)

The correct statements are 2, 3 and 4.

Solution 8.r)

The right answer is number 3. In fact, the `print()` method, which takes a `double` as input, of the superclass `PrintNumber` is always called, and this explains the format of the output. The reason why the method in the `PrintInteger` subclass is not called in virtual mode, is because it is not an override, since the type of parameter is different between the two methods. Since this is not an override, using a superclass reference, the only method that can be called, is precisely that of the superclass.

Solution 8.s)

The only correct answer is number 4.

Solution 8.t)

Overload: since a method is uniquely determined by its signature, in a class (or an interface) it is possible to create multiple methods with the same identifier but with a different list of parameters. In cases like this, we can speak of method overloading.
Override: it allows to rewrite in a subclass a method inherited from a superclass (or interface).
We can obtain an example of a subclass, which implements both concepts, by modifying the classes of exercise 8.r:

```
public class PrintNumber {
    public void print(double number){
        System.out.print(number);
    }
}
```

246

```
public class PrintInteger extends PrintNumber{
    //overload
    public void print(int number) {
        System.out.print(number);
    }
    //override
    public void print(double number) {
        System.out.print(number);
    }
}
```

Solution 8.u)

The correct statements are the numbers 1 and 4.

Statement 2 is not correct because the return type of the `SumInteger` class is covariant (see section 8.2.3.1). Statement 3 is not correct because there is autoboxing-unboxing, which we have already discussed in sections 3.3.2 and 4.3.4.1, and which we will further discuss in sections 12.1.2 and 13.6.

The number 4 is correct because if for example we execute this code:

```
SumInteger sumInteger = new SumInteger();
sumInteger.sum(1.0, 1.0);
```

we will get this exception at runtime:

```
Exception in thread "main" java.lang.ClassCastException:
  java.base/java.lang.Double cannot be cast to
  java.base/java.lang.Integer
    at SumInteger.sum(SumInteger.java:4)
    at SumInteger.main(SumInteger.java:9)
```

> **Much of Chapter 9 is dedicated to exception handling.**

Solution 8.v)

A possible solution could be the following:

```
public class SumInteger extends SumNumber {
    @Override
    public Integer sum(Number n1, Number n2) {
        if (n1 == null || n2 == null) {
            System.out.println("Impossible to sum a null operand, " +
                "retrieving the default value");
```

247

```
                    return Integer.MIN_VALUE;
            } else if (!(n1 instanceof Integer && n2 instanceof Integer)){
                System.out.println("Pass integer variables only, " +
                    "retrieving the default value");
                return Integer.MIN_VALUE;
            }
            return (Integer)n1 + (Integer)n2;
        }
    }
```

By performing the following `main()` method in fact:

```
    public static void main(String args[]) {
        SumInteger sumInteger = new SumInteger();
        sumInteger.sum(1.0, 1.0);
        sumInteger.sum(null, 1.0);
    }
```

we will get the following output and no exceptions:

```
Pass integer variables only, retrieving the default value
Impossible to sum a null operand, retrieving the default value
```

Note that the first check on the nullity of the parameters is necessary because it is not possible to use the cast on a null variable, nor use the + operator.

Solution 8.w)

A **polymorphic parameter** is a parameter of a declared method of a certain type (maybe abstract), but that will point to an instance of its own subclass at runtime.

Heterogeneous collections are collections of different objects, such as an array of Number, which contains objects of its subclasses such as Integer.

A **virtual call to a method** is obtained, when a method is invoked using a reference of a super-class (that can be abstract) that is actually redefined in a subclass.

Solution 8.x)

Let's rewrite the `Contact` and `Special` classes by eliminating the `printDetails()` method, and replace it with the override of the `toString()` method. First, let's also insert the following `toString()` method within the `Entity` class

```
    @Override
    public String toString() {
        return "Id=" + getId();
    }
```

Let's call this method in the `toString()` method of the `Contact` subclass in the following way:

```
    @Override
    public String toString() {
        return super.toString() + "\nName=" + name +
           "\nPhone number=" + phoneNumber + "\nAddress=" + address;
    }
```

So, we can greatly simplify the `toString()` method of the `Special` class:

```
    @Override
    public String toString() {
        return  super.toString() + "\nRingtone=" + ringtone;
    }
```

We could implement the `PhoneBook` class in the following way (changes in bold):

```
package phonebook.data;

public class PhoneBook implements Data {
    private static PhoneBook instance;
    public Contact[] contacts;

    private PhoneBook () {
        contacts = new Contact[]{new Contact("Claudio De Sio Cesari",
          "13, Java Street", "131313131313"),
            new Contact("Stevie Wonder", "10, Music Avenue", "1010101010"),
            new Contact("Gennaro Capuozzo", "1, Four Days of Naples Square",
              "1111111111"),
            new Special("Mario Ruoppolo", "Neruda Street, 3", "333333",
              "The Postman"),
            new Special("Vincenzo Malinconico", "Courts Street, 8", "888888",
              "Tuca Tuca"),
            new Special("Logan Howlett", "Canada Square, 6", "66666", "Hurt")
        };
    }

    public static PhoneBook getInstance() {
        if (instance == null) {
            instance = new PhoneBook();
        }
        return instance;
    }

    public Contact[] getContacts() {
        return contacts;
    }

    public Special[] getSpecialContacts() {
        Special[] specialContactsFound = new Special[contacts.length];
        for (int i = 0, j = 0; i < contacts.length; ++i) {
            System.out.println(contacts[i]);
```

```
            if (contacts[i] == null) {
                break;
            }
            if (contacts[i] instanceof Special) {
                specialContactsFound[j] = (Special)contacts[i];
                j++;
            }
        }
        return specialContactsFound;
    }
}
```

While the User class may undergo small changes (in bold):

```
package phonebook.business;
import phonebook.data.*;

public class User {
    public Contact[] searchContactsByName(String name) {
        Contact[]contacts = PhoneBook.getInstance().getContacts();
        Contact []contactsFound = new Contact[contacts.length];
        for (int i = 0, j = 0;  i < contactsFound.length; i++) {
            if (contacts[i] == null) {
                break;
            }
            if (contacts[i].getName().toUpperCase().contains(name.toUpperCase())) {
                contactsFound[j] = contacts[i];
                j++;
            }
        }
        return contactsFound;
    }

    public Special[] searchSpecialContactsByName(String name) {
        Special[]specialContacts = PhoneBook.getInstance().getSpecialContacts();
        Special[]specialContactsFound = new Special[specialContacts.length];
        for (int i = 0, j = 0;  i < specialContacts.length; i++) {
            if (specialContacts[i] == null) {
                break;
            }
            if (specialContacts[i].getName().toUpperCase().contains(name.toUpperCase())) {
                specialContactsFound[j] = specialContacts[i];
                j++;
            }
        }
        return specialContactsFound;
    }
}
```

Also for the classes `SearchContacts` and `SearchSpecialContacts` the changes are minimal (in bold), and concern the printing of objects using the `toString()` method, instead of the `printDetails()` method. The `SearchContacts` class follows:

```
package phonebook.ui;

import phonebook.data.*;
import phonebook.business.User;
import java.util.Scanner;


public class SearchContacts {
    public static void main(String args[]) {
        System.out.println("Search Contacts");
        System.out.println();
        var user = new User();
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Contact[] foundContacts = user.searchContactsByName(input);
        System.out.println("Contacts found with name containing \"" + input
            + "\"");
        for (Contact contact : foundContacts) {
            if (contact != null) {
                System.out.println(contact + "\n");
            }
        }
    }
}
```

And finally, the `SearchSpecialContacts` class:

```
package phonebook.ui;
import phonebook.data.*;
import phonebook.business.User;
import java.util.Scanner;

public class SearchSpecialContacts {
    public static void main(String args[]) {
        System.out.println("Search Special Contacts");
        System.out.println();
        var user = new User();
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Special[] specialContactsFound =
            user.searchSpecialContactsByName(input);
        System.out.println("Special Contacts found with name containing \"" +
            input + "\"");
```

```
        for (Special special : specialContactsFound) {
            if (special != null) {
                System.out.println(special + "\n");
            }
        }
    }
}
```

Solution 8.y)

We could write the `getOrdinaryContacts()` method in the `PhoneBook` class as follows:

```
    public Contact[] getOrdinaryContacts() {
        Contact[] ordinaryContactsFound = new Contact[contacts.length];

        for (int i = 0, j = 0; i < contacts.length; ++i) {
            if (contacts[i] == null) {
                break;
            }
            if (!(contacts[i] instanceof Special)) {
                ordinaryContactsFound[j] = contacts[i];
                j++;
            }
        }

        return ordinaryContactsFound;
    }
```

While the method looks `searchOrdinaryContactsByName()` in the `User` class could be as follows:

```
    public Contact[] searchOrdinaryContactsByName(String name) {
        Contact[]ordinaryContacts = PhoneBook.getInstance().getOrdinaryContacts();
        Contact[]ordinaryContactsFound = new Contact[ordinaryContacts.length];

        for (int i = 0, j = 0;  i < ordinaryContacts.length; i++) {
            if (ordinaryContacts[i] == null) {
                break;
            }
            if (ordinaryContacts[i].getName().toUpperCase().contains(name.toUpperCase())) {
                System.out.println(ordinaryContacts[i]);
                ordinaryContactsFound[j] = ordinaryContacts[i];
                j++;
            }
        }

        return ordinaryContactsFound;
    }
```

Finally, we could implement the `SearchOrdinaryContacts` class as follows:

```
package phonebook.ui;
import phonebook.data.*;
import phonebook.business.User;
import java.util.Scanner;

public class SearchOrdinaryContacts {
    public static void main(String args[]) {
        System.out.println("Search Ordinary Contacts");
        System.out.println();
        var user = new User();
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Contact[] ordinaryContactsFound = user.searchOrdinaryContactsByName(input);
        System.out.println("Ordinary Contacts found with name containing \"" +
          input + "\"");
        for (Contact ordinary : ordinaryContactsFound) {
            if (ordinary != null) {
                System.out.println(ordinary + "\n");
            }
        }
    }
}
```

Solution 8.z)

We could implement the required `ContactFactory` class, creating an overload for the method called `getContact()`:

```
package phonebook.data;

import phonebook.data.Special;
import phonebook.data.Contact;

public class ContactFactory {

    public static Contact getContact(String name, String phoneNumber,
                                                 String address) {
        return new Contact(name, phoneNumber, address);
    }

    public static Special getContact(String name, String phoneNumber,
                                        String address, String ringtone) {
        return new Special(name, phoneNumber, address, ringtone);
    }
}
```

We have changed the constructor of the PhoneBook class as follows (changes in bold):

```
    private PhoneBook () {
        contacts = new Contact[] {
            new Contact("Claudio De Sio Cesari","13, Java Street","131313131313"),
            new Contact("Stevie Wonder","10, Music Avenue","1010101010"),
            new Contact("Gennaro Capuozzo",
               "1, Four Days of Naples Square","1111111111"),
            new Special("Mario Ruoppolo","Neruda Street, 3","333333",
               "The Postman"),
            new Special("Vincenzo Malinconico","Courts Street, 8","888888",
               "Tuca Tuca"),
            new Special("Logan Howlett","Canada Square, 6","66666","Hurt"),
            null,
            null,
            null,
            null
        } ;
    }
```

We have modified the User class, not only by adding the required add() method, but also by introducing the phoneBook instance variable, and a constructor that initializes it (changes in bold):

```
package phonebook.business;
import phonebook.data.*;

public class User {
    private PhoneBook phoneBook;

    public User() {
        phoneBook = PhoneBook.getInstance();
    }

    public Contact[] searchContactsByName(String name) {
        Contact[]contacts = phoneBook.getContacts();
        Contact[]contactsFound = new Contact[contacts.length];
        for (int i = 0, j = 0;  i < contacts.length; i++) {
            if (contacts[i] == null) {
                break;
            }
            if (contacts[i].getName().toUpperCase().
                                        contains(name.toUpperCase())) {
                contactsFound[j] = contacts[i];
                j++;
            }
        }
        return contactsFound;
    }
```

```java
    public Special[] searchSpecialContactsByName(String name) {
        Special[]specialContacts = phoneBook.getSpecialContacts();
        Special[]specialContactsFound = new Special[specialContacts.length];

        for (int i = 0, j = 0;  i < specialContacts.length; i++) {
            if (specialContacts[i] == null) {
                break;
            }
            if (specialContacts[i].getName().toUpperCase().
                                          contains(name.toUpperCase())) {
                specialContactsFound[j] = specialContacts[i];
                j++;
            }
        }

        return specialContactsFound;
    }

    public Contact[] searchOrdinaryContactsByName(String name) {
        Contact[]ordinaryContacts = phoneBook.getOrdinaryContacts();
        Contact[]ordinaryContactsFound = new Contact[ordinaryContacts.length];

        for (int i = 0, j = 0;  i < ordinaryContacts.length; i++) {
            if (ordinaryContacts[i] == null) {
                break;
            }
            if (ordinaryContacts[i].getName().toUpperCase().
                                          contains(name.toUpperCase())) {
                System.out.println(ordinaryContacts[i]);
                ordinaryContactsFound[j] = ordinaryContacts[i];
                j++;
            }
        }

        return ordinaryContactsFound;
    }

    public void add(Contact contactToAdd) {
        Contact[] contacts = phoneBook.getContacts();
        for (int i = 0; i < contacts.length; ++i) {
            if (contacts[i] == null) {
                contacts[i] = contactToAdd;
                break;
            }
        }
    }
}
```

In fact, if we had not also created the instance variable and the constructor, we could have had

an unexpected situation regarding the class SearchContacts, which we decided to modify it in the following way (changes in bold):

```
package phonebook.ui;

import phonebook.data.*;
import phonebook.business.User;
import java.util.Scanner;

public class SearchContacts {
    public static void main(String args[]) {
        System.out.println("Search Contacts");
        System.out.println();
        var user = new User();
        Contact newContact = ContactFactory.getContact(
          "Molly Malone", "123456789", "Suffolk St, Dublin 2, D02 KX03, Ireland");
        Special specialContact = ContactFactory.getContact(
          "Phil Lynott", "987654321", "Harry St, Dublin, Ireland", "Rosalie");
        user.add(newContact);
        user.add(specialContact);
        System.out.println("Enter name or part of the name to be searched");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        Contact[] foundContacts = user.searchContactsByName(input);
        System.out.println("Contacts found with name containing \"" + input
          + "\"" );
        for (Contact contact : foundContacts) {
            if (contact != null) {
                System.out.println(contact + "\n");
            }
        }
    }
}
```

If we had not introduced the variable and the constructor in the PhoneBook class, the objects instantiated in the main() method newContact and specialContact, would have had id = 1 and id = 2 respectively. In fact, the getSerialNumber() method of the Counter class, called by the constructor of the Contact class, would have been called first for these two instances, while the getInstance() method of the PhoneBook class, which in turn would have initialized the contact array, would have been called later. Therefore, we would have had that id of the original contacts created in the constructor of the PhoneBook class, would have started from number 3.

Chapter 9
Exercises

Exceptions and assertions

Exception handling is a key topic, it is very important to learn every detail (they are not so many). The assertions are much less used, but could be used profitably.

1. Any exception that extends `ArithmeticException` is an unchecked exception.

2. An `Error` differs from an `Exception` because it cannot be launched; in fact, it does not extend the `Throwable` class.

3. The following code:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Division by zero...");
}
catch (NullPointerException exc) {
    System.out.println("Null reference...");
}
catch (Exception exc) {
    System.out.println("Generic exception...");
}
```

```
finally {
    System.out.println("Finally!");
}
```

will produce the following output:

```
Division by zero...
Generic exception...
Finally!
```

4. The following code:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    System.out.println("Generic exception...");
}
catch (ArithmeticException exc) {
    System.out.println("Division by zero...");
}
catch (NullPointerException exc) {
    System.out.println("Null reference...");
}
finally {
    System.out.println("Finally!");
}
```

will cause an exception at runtime.

5. The throw keyword allows you to "throw" only the subclasses of Exception that are defined by the programmer.

6. The throw keyword allows you to "throw" only the subclasses of Exception.

7. If a method uses the throw keyword, in the same method the exception to be thrown must be handled, or the method itself must use a throws clause.

8. The Error class cannot be extended.

9. If a m2() method overrides a m2() method inherited from the superclass, it can declare new exceptions with the throws clause, only if these are subclasses with respect to those declared by the throws clause of the m2() method of the superclass.

10. From version 1.4 of Java it is possible to "wrap" another exception in an exception.

Exercise 9.b) Exception Handling, True or False:

1. If an assertion is not verified in an application, we must talk about bugs.

2. An assertion that is not verified causes the JVM to launch an `AssertionError`.

3. The preconditions serve to test the correctness of the parameters of public methods.

4. The use of assertions is not recommended to test the correctness of data entered by a user.

5. A postcondition is useful to verify that an assertion is verified at the end of a method.

6. An internal invariant allows to test the correctness of the flows within the methods.

7. A class invariant is a particular internal invariant that must be verified for all instances of a certain class, at any time of their life cycle, except during the execution of some methods.

8. An invariant on the execution flow, it is usually an assertion with a syntax like: `assert false;`

9. It is not possible to compile a program that makes use of assertions with the JDK 1.3.

10. It is not possible to run a program that makes use of assertions with the JDK 1.3.

Exercise 9.c)

Consider the classes created in the exercises of Chapter 8: `File`, `SourceFile`, `Editor`, `IDE` and `JavaIDE`. We also consider the `addText(String)` method of the `SourceFile` class we created in exercise 8.f, which we coded in the following way:

```
public void addText(String text) {
    if (content == null) {
        content ="";
    }
    if (text != null) {
        content += text;
    }
}
```

The control of the `if` clause can certainly be improved. What would you use in this case, assertions or exceptions?

Exercise 9.d)

> **(SPOILER ALERT: next lines will reveal the solution of the previous exercise!)**

After reading the solution of the previous exercise, use the keywords throw and (possibly) throws to handle any exceptions in the addText(String) method mentioned in the previous exercise.

Exercise 9.e)

Now let's consider the method addText(String, int) that we created in exercise 8.g and that we coded in the following way:

```
public void addText(String text, int position) {
    final int length = content.length();
    if (content != null && text != null && position > 0
    && position < length) {
        content = content.substring(0, position) + text +
        content.substring(position);
    }
}
```

Handle an exception (or multiple exceptions) using the try-catch keywords, and possibly also the finally clause.

Exercise 9.f)

After doing the previous exercise, create a SourceFileTest test class to verify that exception handling works correctly.

Exercise 9.g)

Consider the sources created with the exercises of Chapter 6. Create custom exceptions to handle unexpected situations. In particular, create an exception that is triggered in the Purse class constructor when too many coins are specified, call it FullPurseException. Handle the exception in the constructor by instantiating the object with a limited number of elements (i.e. without changing the behavior already defined in the exercise of Chapter 6). Also handle the NullPointerException (which statement could generate this exception?).

260

Consider the sources created with the exercises of Chapter 6. Use custom exception handling to handle unexpected situations. In particular, use the FullPurseException exception to be thrown in the add() method we defined in the following way:

```java
public void add(Coin coin) {
    System.out.println("Let's try adding one " +
                coin.getDescription());
    int freeIndex = firstFreeIndex();
    if (freeIndex == -1) {
        System.out.println("Purse full! The coin " +
                coin.getDescription() + " has not been added!");
    } else {
        coins[freeIndex] = coin;
        System.out.println(coin.getDescription() + " has been added");
    }
}
```

Let the FullPurseException exception be thrown appropriately. Also handle any other exceptions.

Consider the sources created with the exercises of Chapter 6. Create custom exceptions to handle unexpected situations. In particular, use the CoinNotFoundException exception to be thrown in the withdraw() method that we defined in the following way:

```java
public Coin withdraw(Coin coin) {
    System.out.println("Let's try to get a "  +
        coin.getDescription());
    Coin foundCoin = null;
    int foundCoinIndex = foundCoinIndex(coin);
    if (foundCoinIndex == -1) {
        System.out.println("Coin not found!");
    } else {
        foundCoin = coin;
        coins[foundCoinIndex] = null;
        System.out.println("One " + coin.getDescription() + " withdraw");
    }
    return foundCoin;
}
```

Let the CoinNotFoundException exception be thrown appropriately. Also handle any other exceptions.

Exercise 9.l)

Let's consider the sources created with the exercises created of Chapter 6. Modify the `CoinsTest` class to correctly handle the `FullPurseException` exception.

Exercise 9.m)

Add assertions in the `Purse` class constructor.

Exercise 9.n)

Which of the following statements are correct?

1. The `RuntimeException` are unchecked exception.

2. `ArithmeticException` is a checked exception.

3. `ClassCastException` is an unchecked exception.

4. `NullPointerException` is a checked exception.

Exercise 9.o)

Which of the following statements are correct?

1. In the `throws` clause it is possible to declare only checked exceptions.

2. In the `throws` clause it is possible to declare only unchecked exceptions.

3. In the `throws` clause it is possible to declare a `NullPointerException`.

4. With the `throw` keyword it is possible to throw only checked exceptions.

5. With the `throw` clause it is possible to throw only unchecked exception.

6. The `throws` clause is mandatory if a checked exception could be launched in our method.

7. A method that declares a `throws` clause, can only be invoked if it is handled within a `try-catch` block.

Exercise 9.p)

Which of the following statements are correct?

1. New exceptions can only be defined as checked exception.

2. If we define a subclass of `NullPointerException`, it can be caught as a `NullPointerException`.

3. If we define a subclass of `NullPointerException`, it will be caught instead of the `NullPointerException`.

4. If we define a subclass of `ArithmeticException`, it will be thrown in case there is a problem in an arithmetic operation.

Exercise 9.q)

Not taking into account the `try-with-resources` construct, the `finally` block is mandatory (choose all valid statements):

1. When there are no `catch` blocks after a `try` block.

2. When there are no `try` blocks before a `catch` block.

3. When there are at least two `catch` blocks after a `try` block.

4. Never.

Exercise 9.r)

Considering the following method:

```
public void methodThatThrowsAnException() throws ArrayIndexOutOfBoundsException{
    //INSERT CODE HERE
}
```

Choose from the following snippets those that could be written in the method `methodThatThrowsAnException()` method, so that it is valid:

1. `throw new ArrayIndexOutOfBoundsException();`

2. ```
int i=0, j=0;
    try {
      i = i/j;
    } catch(ArithmeticException e) {
      throw new ArrayIndexOutOfBoundsException ();
    }
```

3. `int i = 0;`

4. `System.out.println();`

**263**

*Exercise 9.s)*

Considering the following class:

```
public class Exercise9S {
    public static void main(String args[]) throws NullPointerException {
        Exercise9S e = new Exercise9S();
        e.method();
    }

    public NullPointerException method() throws NullPointerException  {
        String s = null;
        try {
            s.toString();
        } catch(ArithmeticException e) {
            throw new NullPointerException ();
        }
        return null;
    }
}
```

1. Which of the following statements are correct?

2. The code does not compile because the `method()` method cannot return `NullPointerException`.

3. The code does not compile because the `method()` method returns `null` and not a `NullPointerException`.

4. The code does not compile because the `main()` method does not declare the right exception in its `throws` clause.

5. The code compiles but at runtime ends with a `NullPointerException`.

6. The code compiles but at runtime ends with an `Exception`.

7. The code compiles but at runtime ends with an `ArithmeticException`.

Exercise 9.t)

Create a `SlidingDoor` class that declares two methods, `open()` and `close()`, where the latter must be compatible to be called with the `try-with-resources` technique.

Exercise 9.u)

Considering the solution of the exercise 9.t (i.e. the `SlidingDoor` class): write a simple class that tests its operation by using the `try-with` resource construct.

264

Exercise 9.v)

Let's continue with exercise 8.u where we had verified that the following classes compile with-
out errors, but that `SumInteger` could throw an exception at runtime.

```
public abstract class SumNumber {
    public abstract Number sum(Number n1, Number n2);
}

public class SumInteger extends SumNumber{
    @Override
    public Integer sum(Number n1, Number n2) {
        return (Integer)n1 + (Integer)n2;
    }
}
```

In fact, with the following instructions:

```
SumInteger sumInteger = new SumInteger();
sumInteger.sum(1.0, 1.0);
```

we will get the following exception at runtime:

```
Exception in thread "main" java.lang.ClassCastException:
  java.base/java.lang.Double cannot be cast to
  java.base/java.lang.Integer
      at SumInteger.sum(SumInteger.java:4)
      at SumInteger.sum(SumInteger.java:9)
```

In the exercise 8.v, we asked to make the implementation of the `SumInteger` class robust, and
the result was the following:

```
public class SumInteger extends SumNumber {
    @Override
    public Integer sum(Number n1, Number n2) {
        if (n1 == null || n2 == null) {
            System.out.println("Impossible to sum a null operand, " +
                "retrieving the default value");
            return Integer.MIN_VALUE;
        } else if (!(n1 instanceof Integer && n2 instanceof Integer)) {
            System.out.println("Pass integer variables only, " +
                "retrieving the default value");
            return Integer.MIN_VALUE;
        }
        return (Integer)n1 + (Integer)n2;
    }
}
```

Now that we know the theory of exceptions, redesign the summarized class using exception
handling.

Exercise 9.z)

Create a simple test class for the `SumInteger` class that we created in exercise 9.v.

Chapter 9
Exercise Solutions

Exceptions and assertions

1. **True**, because `ArithmeticException` is subclass of `RuntimeException`.

2. **False**.

3. **False**, will produce the following output:

```
Division by zero...
Finally!
```

4. **False**, will produce a compile-time error (the order of the `catch` blocks is not valid).

5. **False**.

6. **False**, only the subclasses of `Throwable`.

7. **True**.

8. **False**.

9. **True**.

10. **True**.

Solution 9.b) Exception Handling, True or False:

1. **True**.

2. **True**.

3. **False**.

4. **True**.

5. **True**.

6. **True**.

7. **True**.

8. **True**.

9. **True**.

10. **True**.

Solution 9.c)

As we saw in section 9.6.3.1, we should never use assertions to test the parameters of a public method. So undoubtedly it is more correct to use exception handling.

Solution 9.d)

A possible implementation could be the following:

```
public void addText(String text)  throws RuntimeException {
   if (content == null) {
        content ="";
   }
   if (text != null) {
        throw new RuntimeException("text = null");
   }
   content += text;
}
```

Note that we have thrown a RuntimeException, but we could have thrown any other exception (e.g. Exception itself). Furthermore, the throws clause next to the method declaration is not technically mandatory, but advisable.

The code could be similar to the following:

```
public void addText(String text, int position) {
    try {
        if (text != null) {
            content = content.substring(0, position) + text +
                    content.substring(position);
        }
    } catch (NullPointerException exc) {
        System.out.println("The content is null : " + exc.getMessage());
        content = "" + text;
    } catch (StringIndexOutOfBoundsException exc) {
        System.out.println("The index " + position + " is not valid : " +
                exc.getMessage());
        content = (position < 0 ? text + content : content + text);
    }
}
```

In this example we have only checked if the text to be added is null, in which case no operation is performed. Then we catch the NullPointerException that would occur in the case the content variable was null. In the catch clause we printed a meaningful message and maintained consistency with the method previously presented.

We also handled a StringIndexOutOfBoundsException that would be thrown if the specified position contained a negative number or a number greater than the size of the file's contents. Also in this case, in the catch clause we first printed a meaningful message, and then implemented a solution. In particular, we have ensured that (also using a ternary operator), if the position variable is specified with a negative value, then the text variable is placed at the beginning before the content variable (as if value 0 had been specified). If instead it is set with a value higher than the last index available for the content, then the text variable is added to the end of the content.

At the end, we can avoid the use of the finally clause.

The code of the SourceFileTest class could be the following:

```
public class SourceFileTest {
    public static void main(String args[]) {
        SourceFile sourceFile = new SourceFile("Test.java",
        FileType.JAVA,"public class MyClass {\n\r");
        System.out.println(sourceFile.getContent());
        // Test addText(String) correct
        sourceFile.addText("}");
```

```
        System.out.println(sourceFile.getContent());
        // Test addText(String,int) correct
        sourceFile.addText("//Test adding text\n\r", 23);
        System.out.println(sourceFile.getContent());
        // Test addText(String,int) incorrect
        sourceFile.addText("//Test adding text\n\r", -1);
        System.out.println(sourceFile.getContent());
        // Test addText(String,int) incorrect
        sourceFile.addText("//Test adding text\n\r", 100);
        System.out.println(sourceFile.getContent());
        SourceFile emptySourceFile = new SourceFile("EmptyFile.c",
          FileType.C);
        emptySourceFile.addText("//Test adding text\n\r", 3);
        System.out.println(emptySourceFile.getContent());
        SourceFile emptySourceFile2 = new SourceFile("EmptyFile2.cpp",
          FileType.C_PLUS_PLUS);
        emptySourceFile2.addText("//Test adding text\n\r");

    }
}
```

Solution 9.g)

The new implementation of the `FullPurseException` class, could be the following:

```
public class FullPurseException extends Exception {
    public FullPurseException (String message) {
        super(message);
    }
}
```

The `Purse` class constructor implementation, with the new requirements could be transformed as follows:

```
    public Purse(int... values) {
        try {
            int numberOfCoins = values.length;
            for (int i = 0; i < numberOfCoins; i++) {
                if (i >= 10) {
                    throw new FullPurseException (
                        "Only the first 10 coins have been inserted!");
                }
                coins[i] = new Coin(values[i]);
        } catch (FullPurseException | NullPointerException exc) {
            System.out.println(exc.getMessage());
        }
    }
```

270

Note that passing `null` as argument to the constructor, we could cause a `NullPointerException` when the `length` variable is used.

With a multi-catch we guaranteed the functioning of the constructor without interrupting the program, printing the message of the problem that could occur. However, it is not correct in this case to manage these two types of exception in the same way, since in the case of `NullPointerException` the printed message will simply be:

```
null
```

which is not very explanatory!

It would be better to handle the two exceptions in the following way:

```java
public Purse(int... values) {
    try {
        int numberOfCoins = values.length;
        for (int i = 0; i < numberOfCoins; i++) {
            if (i >= 10) {
                throw new FullPurseException (
                    "Only the first 10 coins have been inserted!");
            }
            coins[i] = new Coin(values[i]);
        }
//  } catch (FullPurseException | NullPointerException exc) {
    } catch (FullPurseException exc) {
        System.out.println(exc.getMessage());
    } catch (NullPointerException exc) {
        System.out.println("The purse has been created empty");
    }
}
```

Solution 9.h)

The implementation of the `add()` method of the `Purse` class with the new requirements, could be updated as follows:

```java
public void add(Coin coin) throws FullPurseException {
    try {
        System.out.println("Let's try adding one " +
                coin.getDescription());
    } catch (NullPointerException exc) {
        throw new NullPointerException("The added coin was null");
    }
    int freeIndex = firstFreeIndex();
    if (freeIndex == -1) {
        throw new FullPurseException("Purse full! The coin "
                + coin.getDescription() + " has not been added!");
```

```
        } else {
            coins[freeIndex] = coin;
             System.out.println(coin.getDescription() + " has been added");
        }
    }
```

Note that we also handled the `NullPointerException` by capturing it and raising it with a message better than "null". Also in this case, the exception would occur if the argument was `null`, as soon as the `getDescription()` method was called on the `coin` object (which would be `null`).

Solution 9.i)

The requested exception could be the following:

```
public class CoinNotFoundException extends Exception {
    public CoinNotFoundException(String message) {
        super(message);
    }
}
```

The code of the `withdraw()` method of the `Purse` class, with the new requirements, could be updated as follows:

```
    public Coin withdraw(Coin coin) throws CoinNotFoundException {
        try {
        System.out.println("Let's try to get a " +
            coin.getDescription());
        } catch (NullPointerException exc) {
            throw new NullCoinException();
        }
        Coin foundCoin = null;
        int foundCoinIndex = foundCoinIndex(coin);
        if (foundCoinIndex == -1) {
            throw new CoinNotFoundException("Coin not found!");
        } else {
            foundCoin = coin;
            coins[foundCoinIndex] = null;
            System.out.println("One " + coin.getDescription() + " withdrawn");
        }
        return foundCoin;
    }
```

Note that we also handled the `NullPointerException` by capturing it and raising it with a message better than "null". Also in this case, the exception would occur if the argument was null, as soon as the `getDescription()` method is called on the `coin` object (which would be `null`). We can improve this mechanism by creating another custom exception:

272

```java
public class NullCoinException extends RuntimeException {
    public NullCoinException() {
        super("The passed coin was null");
    }
}
```

Note that NullCoinException extends RuntimeException, and therefore it is an unchecked exception that does not need to be declared in the throws clause of the method that launches it. And then the add() method can be changed as follows

```java
public void add(Coin coin) throws FullPurseException {
    try {
        System.out.println("Let's try adding one " +
                coin.getDescription());
    } catch (NullPointerException exc) {
        throw new NullCoinException();
    }
    int freeIndex = firstFreeIndex();
    if (freeIndex == -1) {
        throw new FullPurseException("Purse full! The coin "
                + coin.getDescription() + " has not been added!");
    } else {
        coins[freeIndex] = coin;
        System.out.println(coin.getDescription() + " has been added");
    }
}
```

Solution 9.l)

As the previous exercise, the required code could be the following:

```java
/**
 * Test classe for the Coin and Purse classes.
 *
 * @author Claudio De Sio Cesari
*/
public class CoinsTest {

    public static void main(String args[]) {

        Coin twentyCentsCoin = new Coin(20);
        Coin oneCentCoin = new Coin(1);
        Coin oneEuroCoin = new Coin(100);
        // Creation of a Purse with 11 coins
        Purse purseToFail = new Purse(2, 5, 100, 10, 50, 10, 100, 200, 10, 5, 2);
        // Creation of a Purse with 8 coins
        Purse purse = new Purse(2, 5, 100, 10, 50, 10, 100, 200);
        purse.state();
```

```
try {
    // we add a 20 cents coin
    purse.add(twentyCentsCoin);
} catch (FullPurseException | NullCoinException exc) {
    System.out.println(exc.getMessage());
}

try {
    // we add a 1 cents coin
    purse.add(oneCentCoin);
} catch (FullPurseException | NullCoinException exc) {
    System.out.println(exc.getMessage());
}

try {
    // We add the eleventh coin (we should get an error and the
    // coin will not be added)
    purse.add(oneEuroCoin);
} catch (FullPurseException | NullCoinException exc) {
    System.out.println(exc.getMessage());
}

// We evaluate the status of the purse
purse.state();

try {
    // we withdraw 20 cents
    purse.withdraw(twentyCentsCoin);
} catch (CoinNotFoundException exc) {
    System.out.println(exc.getMessage());
}

try {
    // Let's add the tenth coin again
    purse.add(oneEuroCoin);
} catch (FullPurseException | NullCoinException exc) {
    System.out.println(exc.getMessage());
}

// We evaluate the status of the purse
purse.state();

try {
    // We withdraw a non-existent currency (we should get an error)
    purse.withdraw(new Coin(7));
} catch (CoinNotFoundException exc) {
    System.out.println(exc.getMessage());
}
```

```
        try {
            //We try to add null
            purse.add(null);
        } catch (FullPurseException | NullCoinException exc) {
            System.out.println(exc.getMessage());
        }

        try {
            //We try to withdraw null
            purse.withdraw(null);
        } catch (CoinNotFoundException | NullCoinException exc) {
            System.out.println(exc.getMessage());
        }
        //we test the passage of the null value to the purse constructor
        Purse purseWithNullPointerException = new Purse(null);
        purse.state();
    }
}
```

whose output will be:

```
Created a coin of 20 cents of EURO
Created a coin of 1 cent of EURO
Created a coin of 1 EURO
Created a coin of 2 cents of EURO
Created a coin of 5 cents of EURO
Created a coin of 1 EURO
Created a coin of 10 cents of EURO
Created a coin of 50 cents of EURO
Created a coin of 10 cents of EURO
Created a coin of 1 EURO
Created a coin of 2 EURO
Created a coin of 10 cents of EURO
Created a coin of 5 cents of EURO
Only the first 10 coins have been inserted!
Created a coin of 2 cents of EURO
Created a coin of 5 cents of EURO
Created a coin of 1 EURO
Created a coin of 10 cents of EURO
Created a coin of 50 cents of EURO
Created a coin of 10 cents of EURO
Created a coin of 1 EURO
Created a coin of 2 EURO
The purse contains:
One coin of 2 cents of EURO
One coin of 5 cents of EURO
One coin of 1 EURO
One coin of 10 cents of EURO
One coin of 50 cents of EURO
One coin of 10 cents of EURO
One coin of 1 EURO
One coin of 2 EURO
```

```
Let's try adding one coin of 20 cents of EURO
coin of 20 cents of EURO has been added
Let's try adding one coin of 1 cent of EURO
coin of 1 cent of EURO has been added
Let's try adding one coin of 1 EURO
Purse full! The coin coin of 1 EURO has not been added!
The purse contains:
One coin of 2 cents of EURO
One coin of 5 cents of EURO
One coin of 1 EURO
One coin of 10 cents of EURO
One coin of 50 cents of EURO
One coin of 10 cents of EURO
One coin of 1 EURO
One coin of 2 EURO
One coin of 20 cents of EURO
One coin of 1 cent of EURO
Let's try to get a coin of 20 cents of EURO
One coin of 20 cents of EURO withdrawn
Let's try adding one coin of 1 EURO
coin of 1 EURO has been added
The purse contains:
One coin of 2 cents of EURO
One coin of 5 cents of EURO
One coin of 1 EURO
One coin of 10 cents of EURO
One coin of 50 cents of EURO
One coin of 10 cents of EURO
One coin of 1 EURO
One coin of 2 EURO
One coin of 1 EURO
One coin of 1 cent of EURO
Created a coin of 7 cents of EURO
Let's try to get a coin of 7 cents of EURO
Coin not found!
The passed coin was nulll
The passed coin was nulll
The purse has been created empty
The purse contains:
One coin of 2 cents of EURO
One coin of 5 cents of EURO
One coin of 1 EURO
One coin of 10 cents of EURO
One coin of 50 cents of EURO
One coin of 10 cents of EURO
One coin of 1 EURO
One coin of 2 EURO
One coin of 1 EURO
One coin of 1 cent of EURO
```

Solution 9.m)

We could modify the Purse code in the following way:

```
public class Purse {

    private static final int DIMENSION = 10;
    private final Coin[] coins = new Coin[DIMENSION];

    public Purse(int... values) {
        assert coins.length == DIMENSION;
        try {
            int numberOfCoins = values.length;
            for (int i = 0; i < numberOfCoins; i++) {
                if (i >= 10) {
                    throw new FullPurseException (
                        "Only the first 10 coins have been inserted!");
                }
                coins[i] = new Coin(values[i]);
            }
//  } catch (FullPurseException | NullPointerException exc) {
        } catch (FullPurseException exc) {
            System.out.println(exc.getMessage());
        } catch (NullPointerException exc) {
            System.out.println("The purse has been created empty");
        }
        assert coins.length == DIMENSION;
    }
//...
```

Note that we have defined the constant DIMENSION initialized to 10. Then, as assertions we used a pre-condition and a post-condition. In both cases we have stated the same concept: the length of the coins array is equal to the DIMENSION value. Note also that with these simple assertions we are reinforcing the logic of the constructor, it is as if we were saying "whatever happens, the value of the length of the array cannot change". In this way our code will remain consistent with this assertion through all the changes that will be made later.

Solution 9.n)

Only statement number 3 is correct.

Solution 9.o)

The statements number 3 and number 6 are correct. In fact, the throws clause and the throw command can be used for every type of exception, and this excludes that the statements 1, 2, 4 and 5 are correct. Statement 7 is incorrect because the method that invokes a method that declares a throws clause, could in turn declare the throws clause!

277

Solution 9.p)

None of the statements is correct.

Solution 9.q)

Only the first statement is correct.

Solution 9.r)

All snippets are valid because, in the `throws` clause, the `ArrayIndexOutOfBoundsException` is declared which is an unchecked exception, and which is therefore not explicitly mandatory. Note that answer 2 throws an `ArithmeticException`, handles it and raises an `ArrayIndexOutOfBoundsException`.

Solution 9.s)

The only correct statement is 3. Its correctness excludes the correctness of statements 4, 5 and 6. The 1 and 2 are false because `NullPointerException` is still a class.

Solution 9.t)

A simple implementation could be the following:

```java
public class SlidingDoor implements AutoCloseable {
    public void close(){
        System.out.println("The door is closing");
    }

    public void open(){
        System.out.println("The door is opening");
    }
}
```

Solution 9.u)

The solution could be as simple as the following:

```java
public class Exercise9U {
    public static void main(String args[]) {
        try (SlidingDoor slidingDoor = new SlidingDoor();) {
            slidingDoor.open();
        }
    }
}
```

278

whose output, once executed, will be:

```
The door is opening
The door is closing
```

Solution 9.v)

The solution with exception handling is undoubtedly simpler and more elegant:

```java
public class SumInteger extends SumNumber {
    @Override
    public Integer sum(Number n1, Number n2) {
        Integer result = null;
        try {
            result = (Integer)n1 + (Integer)n2;
        } catch (NullPointerException e) {
            System.out.println("Impossible to sum a null operand");
        } catch (ClassCastException e) {
            System.out.println("Pass only integer variables");
        }
        return result;
    }
}
```

Solution 9.z)

With the following test class:

```java
public class Exercise9Z {
    private static final String FIRST_PART_OF_THE_STRING ="The result is ";
    public static void main(String args[]) {
        SumInteger sumInteger = new SumInteger();
        System.out.println(FIRST_PART_OF_THE_STRING + sumInteger.sum(1.0, 1.0));
        System.out.println(FIRST_PART_OF_THE_STRING + sumInteger.sum(1, null));
        System.out.println(FIRST_PART_OF_THE_STRING + sumInteger.sum(1, 25));
    }
}
```

we can verify the expected results.

Chapter 10 Exercises

A Guided Example to Object-Oriented Programming

For this chapter we have reserved summary exercises about all the topics that we have studied so far. All exercises include programming. In particular, in the first part we will create an application starting from the `Point` class that we defined in the Chapter 10, up to graphically drawing polylines on a Cartesian plane. Then we will create an application that is capable of converting the values expressed in the Kelvin, Celsius and Fahrenheit scale. Finally, we will create an application that simulates a configurable and multi-user text game. From the next chapter on, we will study more complex topics starting the third part about the "Advanced language features", so this is the time to reinforce our programming bases.

Exercise 10.a)

Create:

1. The `Point` class with x and y coordinates, the setter and getter methods, a constructor and a `toString()` method (which is a method inherited from the `Object` class, and therefore inherited in all classes).

2. The `Ruler` class declaring a method `getDistance()` which takes two objects of type `Point` as input, and returns the value of the geometric distance as `double` type. Read the documentation and take advantage of the methods of the `Math` class to perform the calculations. Also consider whether to declare the method as `static` or not.

3. Create an `Exercise10A` class, which prints the distance of two points and check if the result is correct, using an assertion as seen in chapter 9.

> **In order to use the assertions, you must enable them at the time of execution using the `-ea` option (see Section 9.4.2). If you use EJE, assertions should already be enabled by default. You can check inside the EJE options as shown in figure 10.a.1. If you use another IDE instead, please consult the IDE documentation.**
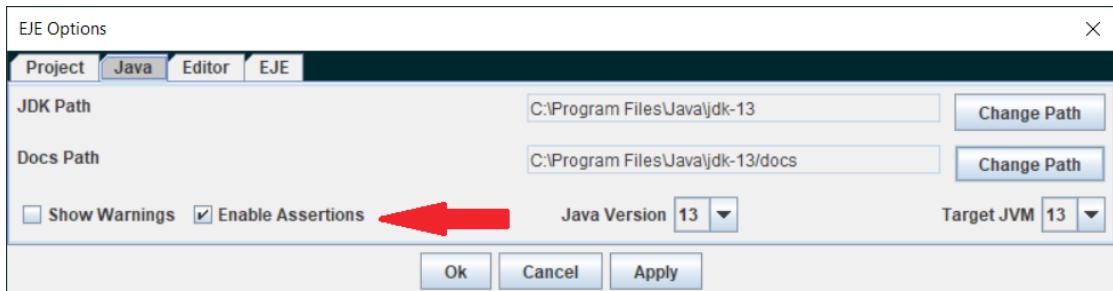
Figure 10.a.1 – Assertions enabling in EJE.

Exercise 10.b)

Starting from the previous exercise, create the `Exercise10B` class. This time we parameterize the application using the array of strings `args`, the input parameter of the `main()` method. In particular, when we launch the `Exercise10B` class, we need to enter the four coordinates of the two points. For example:

```
java Exercise10B 2 1 1 2
```

Since these variables within the `main()` method will be of the `String` type, using the documentation of the `Integer` class, find the correct method that will allow us to transform a string into an integer, and manage any exception.

> **The check made with the assertions in the previous exercise is not required, because we cannot know in advance the values of the coordinates that will be passed as input to the `main()` method.**

Exercise 10.c)

Starting from the solution of the previous exercise, create the `Exercise10C` class using the `Scanner` class, to acquire the coordinates of the two points from the command line. This class must:

1. Request the insertion of the coordinates (one at a time) to the user.

2. Acquire the coordinates using the most appropriate `Scanner` class method (consult the documentation).

3. If the user enters a coordinate in an incorrect format (remember that the coordinates must be integer numbers), the program must manage the exception, print an error message and terminate.

Below is an example of the output that the application should generate:

```
Enter the x coordinate of the first point
2
Enter the y coordinate of the first point
1
Point created:(2,1)
Enter the x coordinate of the second point
1
Enter the y coordinate of the second point
2
Point created:(1,2)
Distance between two points: (2,1) and (1,2) = 1.4142135623730951
```

Exercise 10.d)

Starting from the solution of the previous exercise, create the `Exercise10d` class, performing a refactoring on the `Exercise10C` class, trying to limit the duplicate code and make it more readable.

Exercise 10.e)

Starting from the solution of the previous exercise, create the class `Exercise10E`, making sure that, if the user enters a coordinate with a wrong format (for example by entering a letter), the application must report the error to allow the user to re-enter the coordinate. The program must end only when all the coordinates are entered correctly and the distance is printed. Below is an example of output that could generate the application:

```
Enter the coordinate x of point 1
```

283

```
f
Invalid coordinate f! All coordinates must be integers!
 Please enter only integers
2
Enter the coordinate y of point 1
1
Point created: (2,1)
Enter the coordinate x of point 2
1
Enter the coordinate y of point 2
y
Invalid coordinate y! All coordinates must be integers!
 Please enter only integers
foo
Invalid coordinate foo! All coordinates must be integers!
 Please enter only integers
3
Point created: (1,3)
Distance between two points: (2,1) and (1,3) = 2.23606797749979
```

Exercise 10.f)

Starting from the solution of the previous exercise:

1. create the Segment class, which abstracts the concept of a straight line between two points. This class must declare the two extreme points and the length as instance variables. Also create the setter and getter methods you think you need to implement, a constructor and a toString() method.

2. Create the Exercise10F class, similar to the Exercise10E class, which asks for the coordinates of the two extreme points of the segment and prints its length.

Below is an example of output that could generate the application:

```
Please define a segment specifying its extreme points:
Edit coordinate x of the extreme point 1
2
Edit coordinate y of the extreme point 1
foo
Invalid coordinate foo! All coordinates must be integers!
 Please enter only integers
3
Point created: (2,3)
Edit coordinate x of the extreme point 2
1
Edit coordinate y of the extreme point 2
1
Point created: (1,1)
Segment from P1(2,3) to P2(1,1) with length = 2.23606797749979
```

284

Exercise 10.g)

In this exercise we will use a class called `CartesianPlane`, which will allow us to view our points on a Cartesian plane.

Unfortunately, unless you have already studied all the chapters in this book, you shouldn't be able to understand all the code. However, for the sake of completeness, we report the following code (in any case, find the class ready in the exercises file in the folder **10.g**):

```java
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Dimension;
import java.util.*;

public class CartesianPlane {
    private final static int LENGTH = 600;
    private final static int MID_LENGTH = LENGTH/2;
    private final static int GAP = LENGTH/10;
    private JFrame frame;
    private JPanel panel;
    private ArrayList<Point> points;

    public CartesianPlane() {
        frame = new JFrame("Cartesian Plane");
        panel = new CartesianPlanePanel();
        points = new ArrayList<>();
        setup();
        addDetails();
    }

    private void setup() {
        frame.add(panel);
    }

    private void addDetails() {
        frame.getContentPane().setPreferredSize(new Dimension(LENGTH , LENGTH));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    public void drawPoint(Point point) {
        points.add(point);
```

```
        panel.repaint();
    }

    private class CartesianPlanePanel extends JPanel {
        public CartesianPlanePanel() {
            setBackground(Color.lightGray);
        }

        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            drawAxes(g);
            g.setColor(Color.RED);
            drawPoints(g);
        }

        private void drawPoints(Graphics g) {
            for (Point point : points) {
                drawPoint(point, g);
            }
        }

        private void drawPoint(Point point, Graphics g) {
            int x = getX(point.getX(), 2);
            int y = getY(point.getY(), 2);
            g.drawOval(x,y,5,5);
            g.fillOval(x,y,5,5);
            g.drawString(point.toString(), x-15, y-8);
        }

        private int getX(int x, int delta) {
            return MID_LENGTH + (gap(x)-delta);
        }

        private int getY(int y, int delta) {
            return MID_LENGTH - (gap(y)+delta);
        }

        private void drawAxes(Graphics g) {
            drawReferencePoints(g);
            g.setColor(Color.BLACK);
            g.drawLine(MID_LENGTH, 0, MID_LENGTH, LENGTH);
            g.drawLine(0, MID_LENGTH, LENGTH, MID_LENGTH);
        }

        private void drawReferencePoints(Graphics g) {
            g.setColor(Color.YELLOW);
            for(int i = 1; i < 10; i++) {
                g.drawLine(GAP*i, 0, GAP*i, LENGTH);
            }
```

286

Figure 10.a.2 – The application in action.

```
        for(int i = 1; i < 10; i++) {
            g.drawLine(0, GAP*i, LENGTH, GAP*i);
        }
    }

    private int gap(int value) {
        return value*GAP;
    }
}

public static void main(String args[]) {
    SwingUtilities.invokeLater(() -> new CartesianPlane());
}
}
```

This is the first programming exercise where you will see an application that uses graphics (the last two chapters of the book are dedicated to this topic). In any case, the purpose of our exercise is not to understand the CartesianPlane class code, but to use this class by creating the Exercise10G class starting from the Exercise10F class. The Exercise10G class will have to open the window containing the Cartesian plane with the following instruction:

```
SwingUtilities.invokeLater(() -> cartesianPlane = new CartesianPlane());
```

(also import javax.swing.SwingUtilities).

Also, will have to:

1. add a new instance variable name to the Point class, with its setter and getter methods and a new constructor.

2. Build a point with the coordinates entered by the user. Also set the name of the point, with an automatic mechanism (for example the first point must be called P1, the second P2, the third P3, and so on).

3. Instead of printing the point details on the console, the class must invoke the drawPoint() method of the CartesianPlane class.

4. Check that the points are correctly displayed on the Cartesian plane (see Figure 10.a.2).

Exercise 10.h)

Starting from the previous exercise, evolve the Exercise10G class into the Exercise10H class. This must continue to ask the user to enter the coordinates of a point indefinitely, until the application is closed by closing the Cartesian plane window. In this way the user can view multiple points on the Cartesian axis.

> **The resolution of this exercise is very simple.**

Exercise 10.i)

Starting from the previous exercise, and using the Segment class created in Exercise 10.f, we provide a new version of the CartesianPlane class, to which we have added the functionality of being able to display segments (just invoke the drawSegment() method). The new CartesianPlane class is shown below with the changes in bold (however, find the class ready in the exercises file in folder **10.i**):

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Dimension;
import java.util.*;
```

288

```java
public class CartesianPlane {
    private final static int LENGTH = 600;
    private final static int MID_LENGTH = LENGTH/2;
    private final static int GAP = LENGTH/10;
    private JFrame frame;
    private JPanel panel;
    private ArrayList<Point> points;
    private ArrayList<Segment> segments;

    public CartesianPlane() {
        frame = new JFrame("Cartesian Plane");
        panel = new CartesianPlanePanel();
        points = new ArrayList<>();
        segments = new ArrayList<>();
        setup();
        addDetails();
    }

    private void setup() {
        frame.add(panel);
    }

    private void addDetails() {
        frame.getContentPane().setPreferredSize(new Dimension(LENGTH , LENGTH));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    public void drawPoint(Point point) {
        points.add(point);
        panel.repaint();
    }

    public void drawSegment(Segment segment) {
        segments.add(segment);
        panel.repaint();
    }

    private class CartesianPlanePanel extends JPanel {
        public CartesianPlanePanel() {
            setBackground(Color.lightGray);
        }

        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            drawAxes(g);
            g.setColor(Color.RED);
```

```
        drawPoints(g);
        g.setColor(Color.BLUE);
        drawSegments(g);
    }

    private void drawSegments(Graphics g) {
        for (Segment segment : segments) {
            Point extremePoint1 = segment.getExtremePoint1();
            drawPoint(extremePoint1, g);
            Point extremePoint2 = segment.getExtremePoint2();
            drawPoint(extremePoint2, g);
            g.setColor(Color.BLUE);
            g.drawLine(getX(extremePoint1.getX()),
              getY(extremePoint1.getY()), getX(extremePoint2.getX()),
              getY(extremePoint2.getY()));
        }
    }

    private void drawPoints(Graphics g) {
        for (Point point : points) {
            drawPoint(point, g);
        }
    }

    private void drawPoint(Point point, Graphics g) {
        int x = getX(point.getX(), 2);
        int y = getY(point.getY(), 2);
        g.drawOval(x,y,5,5);
        g.fillOval(x,y,5,5);
        g.drawString(point.toString(), x-15, y-8);
    }

    private int getX(int x) {
        return getX(x, 0);
    }

    private int getY(int y) {
        return getY(y, 0);
    }

    private int getX(int x, int delta) {
        return MID_LENGTH + (gap(x)-delta);
    }

    private int getY(int y, int delta) {
        return MID_LENGTH - (gap(y)+delta);
    }

    private void drawAxes(Graphics g) {
```

290

```
            drawReferencePoints(g);
            g.setColor(Color.BLACK);
            g.drawLine(MID_LENGTH, 0, MID_LENGTH, LENGTH);
            g.drawLine(0, MID_LENGTH, LENGTH, MID_LENGTH);
        }

        private void drawReferencePoints(Graphics g) {
            g.setColor(Color.YELLOW);
            for(int i = 1; i < 10; i++) {
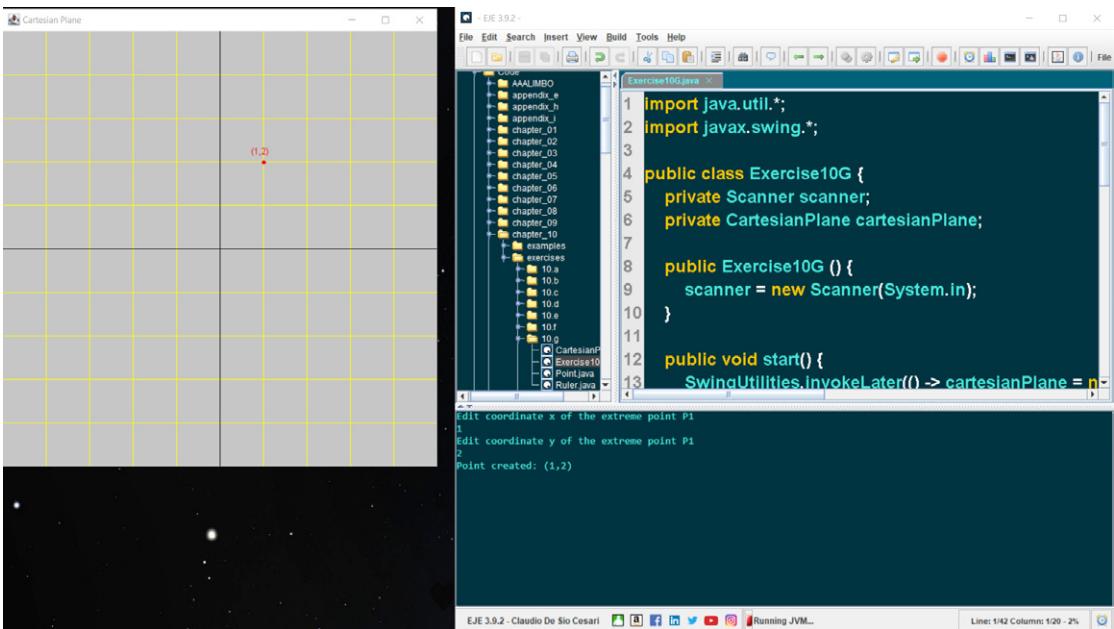                g.drawLine(GAP*i, 0, GAP*i, LENGTH);
            }
            for(int i = 1; i < 10; i++) {
                g.drawLine(0, GAP*i, LENGTH, GAP*i);
            }
        }

        private int gap(int value) {
            return value*GAP;
        }
    }

    public static void main(String args[]) {
        SwingUtilities.invokeLater(() -> new CartesianPlane());
    }
}
```

Create an `Exercise10I` class, similar to the `Exercise10H` class, that allows the user to specify the extremes points of segments that must be displayed on the Cartesian plane.

Exercise 10.j)

Starting from the solution of the previous exercise, create a `Polyline` class, which abstracts the concept of polyline. This must declare an `ArrayList` of segments. A polyline has the constraint that the second extreme point of a segment must coincide with the first extreme point of the next segment. Note that a polyline always consists of at least two segments. Then create one or more constructors, the `addSegment()` method which allows you to add a segment to the polyline, and the `toString()` method. Handle any problems with exceptions, as studied in Chapter 9.

Also create a test class called `PolylineTest` which will have the following `main()` method:

```
    public static void main(String args[]) {
        testCorrectPolyline();
        testCorrectPolylineWithFourPoints();
        testPolylineWithoutSegments();
        testPolylineWithOneSegment();
        testPolylineWithTwoNonConsecutiveSegmentsInConstructor();
```

```
            testPolylineWithThreeNonConsecutiveSegmentsInConstructor();
            testCorrectPolylineAddingAConsecutiveSegment();
            testInvalidPolylineAddingANonConsecutiveSegment();
    }
```

Implement these methods with assertions (if the test fails, an error must be thrown through an assertion).

Exercise 10.k)

In Section 10.4.2, we ran this test with JUnit:

```java
import org.junit.Assert;
import org.junit.Test;

public class PointTest {
    @Test
    public void testDistanceOnX() {
        Point p1 = new Point(1,1);
        Point p2 = new Point(1,2);
        double distance = p1.distance(p2);
        Assert.assertTrue(distance == 1);
    }

    @Test
    public void testDistanceWithNull() {
        Point p1 = new Point(1,1);
        Point p2 = null;
        double distance = p1.distance(p2);
        Assert.assertTrue(distance == -1);
    }

    @Test
    public void testDistanceFromTheSamePoint() {
        Point p1 = new Point(1,1);
        Point p2 = new Point(1,1);
        double distance = p1.distance(p2);
        Assert.assertTrue(distance == 0);
    }

    @Test
    public void testDistanceFromAThreeDimensionalPoint() {
        Point p1 = new Point(1,1);
        Point p2 = new ThreeDimensionalPoint(1,2,2);
        double distance = p1.distance(p2);
        Assert.assertTrue(distance == -1);
    }
}
```

292

for the `Point` class:

```java
public class Point {
    private int x, y;

    public Point() {
        //Constructor without parameters
    }

    public Point(int x, int y) {
        this.setXY(x, y); //this is optional
        //code reuse
    }

    public void setX(int x) {
        this.x = x; //this is mandatory
    }

    public void setY(int y) {
        this.y = y; //this is mandatory
    }

    public void setXY(int x, int y) {
        this.setX(x); //this is optional
        this.setY(y);
    }

    public int getX() {
        return this.x; //this is optional
    }

    public int getY() {
        return this.y; //this is optional
    }

    public double distance(Point p) {
        //square of the difference of the x of the two points
        int tmp1 = (x - p.x)*(x - p.x);
        //square of the difference of the y of the two points
        int tmp2 = (y - p.y)*(y - p.y);
        //square root of the sum of the two squares
        return Math.sqrt(tmp1 + tmp2);
    }
}
```

and for the `ThreeDimensionalPoint` class:

```java
public class ThreeDimensionalPoint extends Point {
    private int z;
```

```java
    public ThreeDimensionalPoint() {
        //Constructor without parameters
    }

    public ThreeDimensionalPoint(int x, int y, int z) {
        this.setXYZ(x, y, z); //Code reuse
    }

    public void setZ(int z) {
        this.z = z; //this is mandatory
    }

    public void setXYZ(int x, int y, int z) {
        this.setXY(x, y); ///Code reuse
        this.setZ(z); //this is optional
    }

    public int getZ() {
        return this.z; //this is optional
    }

    @Override
    public double distance(Point p) {
        if (p instanceof ThreeDimensionalPoint) {
            //Call to a private method using casting
            return this.calculateDistance((ThreeDimensionalPoint)p);
        }
        else {
            return -1; //distance not valid!
        }
    }

    private double calculateDistance(ThreeDimensionalPoint p1) {
        //square of the difference of the x of the two points
        int tmp1=(getX()-p1.getX())*(getX()-p1.getX());
        //square of the difference of the y of the two points
        int tmp2=(getY()-p1.getY())*(getY()-p1.getY());
        //square of the difference of the z of the two points
        int tmp3=(z-p1.z)*(z-p1.z);
        //square root of the sum of the three squares
        return Math.sqrt(tmp1+tmp2+tmp3);
    }

    /* @Override
    public double distance (ThreeDimensionalPoint p) {
        //square of the difference of the x of the two points
        int tmp1=(getX()-p.getX())*(getX()-p.getX());
        //square of the difference of the y of the two points
        int tmp2=(getY()-p.getY())*(getY()-p.getY());
```

```
        //square of the difference of the z of the two points
        int tmp3=(z-p.z)*(z-p.z);
        //square root of the sum of the three squares
        return Math.sqrt(tmp1+tmp2+tmp3);
    } */


    /* @Override
    public double distance(Point p) {
        if (p instanceof ThreeDimensionalPoint) {
            //square of the difference of the x of the two points
            int tmp1=(getX()-p1.getX())*(getX()-p1.getX());
            //square of the difference of the y of the two points
            int tmp2=(getY()-p1.getY())*(getY()-p1.getY());
            //square of the difference of the z of the two points
            int tmp3=(z-p1.z)*(z-p1.z);
            //square root of the sum of the three squares
            return Math.sqrt(tmp1+tmp2+tmp3);
        }
        else {
            return -1; //distance not valid!
        }
    } */
}
```

achieving failure in 50% of tests. In this exercise we will modify the `PointTest` class, solving the problems highlighted by the test.

> **To make things easier, we recommend using an IDE such as Eclipse which already integrates the JUnit tool.**

Exercise 10.l)

Starting from the solution of Exercise 10.j, transform the `PolylineTest` class, using the JUnit tool instead of the assertions.

> **To make things easier, we recommend using an IDE such as Eclipse which already integrates the JUnit tool.**

Exercise 10.m)

Starting from the solution of Exercise 10.i, and adding the `Polyline` and `InvalidPolylineException` classes defined in Exercise 10.j, create along the lines of the `Exercise10I` class, the `Exercise10M` class that allows the user to create polylines on the Cartesian plane. The application will have to request the user to insert point by point the ends of the segments that make up a polyline (which in the meantime will be displayed by the `CartesianPlane` class). Only for the first segment the first extreme point must be specified, while for the following, it is assumed that the first extreme point coincides with the second extreme point of the previous segment.

Below is an example of the output that will generate our application:

```
Let's define the first two segments of a polyline:
Define the first extreme point of the first segment of the polyline:
Edit coordinate x of the extreme point P1
1
Edit coordinate y of the extreme point P1
2
Point created: (1,2)
Define the second extreme point of the first segment of the polyline:
Edit coordinate x of the extreme point P2
3
Edit coordinate y of the extreme point P2
4
Point created: (3,4)
Define the second extreme point of the second segment of the polyline:
Edit coordinate x of the extreme point P3
5
Edit coordinate y of the extreme point P3
6
Point created: (5,6)
Edit coordinate x of the extreme point P4
3
Edit coordinate y of the extreme point P4
3
Point created: (3,3)
Edit coordinate x of the extreme point P5
```

Exercise 10.n)

By testing the solution of Exercise 10.m, we noticed that by specifying a segment of coincident extreme points, the system draws two points one above the other, therefore we should not have to consider it as a segment. So, manage the exception so that the system does not allow you to create segments with two coincident extreme points.

Exercise 10.o)

Let's create the `Exercise10O` class which will represent a converter from the Kelvin scale to the Celsius and Fahrenheit scales. Our application must ask the user to enter a value in Kelvin, and the program will have to convert the entered value into Celsius and Fahrenheit degrees.
Here is an example of output that our application will generate:

```
Enter a Kelvin value
0
0.0  Kelvin equals:
-273.15 Celsius
-459.66998 Fahrenheit
```

Use the float type to do all the calculations.

> **Recall that the formula for converting Kelvin degrees (which we represent with "K") to Celsius (which we represent with "C") is: 0 K - 273,15 = -273,1 C. Recall that the formula for converting degrees Kelvin in Fahrenheit (which we represent with "F") is: (0 K - 273,15) × 9/5 + 32 = -459,7 F.**

Exercise 10.p)

The output of the previous exercise does not satisfy us, as the use of the `float` type does not allow us to define a unique format. Evolve the `Exercise10O` class to the `Exercise10P` class, using the `BigDecimal` class, so that the values are formatted by specifying only the first two decimal places. You need to study the documentation of the `BigDecimal` class.
Below is an example of output that our application will generate:

```
Enter a Kelvin value
0
0 Kelvin equals:
-273.15 Celsius
-459.67 Fahrenheit
```

Exercise 10.q)

Evolve the `Exercise10P` class into the `Exercise10Q` class, so that the user can start from any scale and get to the other scales.
Below is an example of output that our application will generate:

```
Enter a value in Kelvin (for example 12K), Celsius (25C) or Fahrenheit (451F)
0k
0 Kelvin equals:
-273.15 Celsius
-459.67 Fahrenheit
```

or:

```
Enter a value in Kelvin (for example 12K), Celsius (25C) or Fahrenheit (451F)
25c
25 Celsius equals:
298.15 Kelvin
77.00 Fahrenheit
```

or also:

```
Enter a value in Kelvin (for example 12K), Celsius (25C) or Fahrenheit (451F)
500F
500 Fahrenheit equals:
260.00 Celsius
533.15 Kelvin
```

Exercise 10.r)

Evolve the Exercise10Q class into the Exercise10R class, trying to improve the abstraction of our code. So, let's create a Converter class that will be responsible for performing all the arithmetic conversions. It will therefore declare all the (static) methods that are used to convert one scale into another. For example, here is the declaration of the Kelvin to Fahrenheit conversion method:

```
public static BigDecimal convertKelvinToFahrenheit(BigDecimal kelvin) {
    //your code here
}
```

Obviously the Exercise10R class will now have to use the methods of the Converter class.

Exercise 10.s)

Evolve the Exercise10R class into the Exercise10S class, trying to improve the abstraction of our code. So, let's create a Printer class that will be responsible for printing all the messages. It will then declare all the (static) methods needed to print the messages. For example, here is the declaration of the method that prints the error message when inserting an invalid symbol to choose the scale of the entered value:

298

```
public static void printInvalidScale (String scale) {
    //your code here
}
```

Obviously the `Exercise10S` class will now have to use the methods of the `Printer` class.

In this exercise we will try to use the `java.logging` API for the first time. Unlike what is seen in Section 10.4.3, we are going to configure our logging via the following configuration file (found in the **10.t** folder of the book code):

```
handlers=java.util.logging.FileHandler
.level=ALL
java.util.logging.FileHandler.pattern=./exercise10t.log
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
Exercise10T.level=ALL
```

Where it is specified that the log will be generated within a file in the same folder called **exercise10t.log**. It also specifies that all log messages will be printed at any level (`Exercise10T.level = ALL` statement).

Add the following method only within the `Exercise10T` file (evolution of the `Exercise10S` file):

```
    private static void initLogging() {
        try {
            logManager.readConfiguration(
                new FileInputStream("logging.properties"));
        } catch (IOException exception) {
            LOGGER.log(Level.SEVERE,
                "Problem reading configuration file",exception);
        }
    }
```

and invoke it in the first statement of the `main()` method, so that the configuration via file is correctly performed. Also insert some log instructions with different levels, as you like. Test that everything works correctly launching the application and checking the log file. Test the application several times by specifying in the configuration file, in place of the word `ALL`, other configuration levels such as `SEVERE`, `INFO`, etc. checking that only messages with the log levels allowed by the configuration are printed.

Now, we will implement a simple game after all this math and physics! Our application will ask the user to guess a random number between 1 and 100 counting the

number of attempts.

An example of the output that must be generated by this application is the following:

```
I'm thinking of a number between 1 and 100, guess what!
50
Too high, try again
25
Too high, try again
13
Too high, try again
8
Too high, try again
4
Too high, try again
2
Too low, try again
3
Yes! You guessed it after 7 attempts
```

Create the GuessNumber class which contains a single static method that generates the random number between 1 and the maximum number that is passed to it as input. Here is the method declaration:

```
public static int generateRandomNumber(int max) {
    //your code here
}
```

Then, given the following interface:

```
public interface Game {
    public void init();
    public void start();
    public void play();
    public void end();
}
```

implement it in a class called GuessNumberGame, which implements the interface methods in the most appropriate way. In the same class we can also put the main() method that will start the application.

Exercise 10.v)

The solution of the previous exercise, however, did not take into account the possibility that the user enters invalid data. Then evolve the GuessNumberGame class, in order to manage any incorrect user input.

An example of the output that must be generated by this application is the following:

```
I'm thinking of a number between 1 and 100, guess what!
u
Invalid input 'u'. You can only enter integer numbers, please try again
```

Exercise 10.w)

Let's begin to modify our application to make it a multi-player game. So, let's create the `Player` class that needs to define:

1. the instance variables `name` and `id` (which will be used to uniquely identify a player even in case of coincidence of names).

2. A single constuctor.

3. A simple `toString()` method.

Defining an abstract class `MultiPlayerGame` that implements the `Game` interface defined in Exercise 10.u, and which defines:

1. as an instance variable, an `ArrayList` of `Player` objects called `players`.

2. The `getPlayers()` getter method, which returns the `players` instance variable.

3. The `addPlayer()` and `removePlayer()` methods that add or remove a `Player` type object from the `players` instance variable.

4. A `getPlayer()` method which takes the index of the player as input into the `players` list, and which returns the corresponding `Player` object, or throws a `PlayerException` (to be created too), in cases the index is not valid .

5. A `getPlayer()` method that takes as input the name of a player present in the `players` list, and that returns the corresponding `Player` object, or throws a `PlayerException`, in cases where the player with the specified name is not found , or if there are multiple players with the same name on the list.

6. Create a concrete `MultiPlayerGameImpl` ("Impl" stands for implementation) class that extends `MultiPlayerGame`, and defines a new `printPlayers()` method that prints the details of all players in the `players` list.

7. Finally create a `MultiPlayerGameTest` test class that instantiates an object from `MultiPlayerGameImpl`, and all tests all the methods defined by `MultiPlayerGame`.

301

Exercise 10.x)

Considering the classes implemented in the solution of Exercise 10.w (excluding `MultiPlayerGameImpl`), and the following interface:

```
public interface Configurator {
    public void config();
}
```

The goal of this exercise is to create an implementation of the `Configurator` interface called `MultiPlayerGameConfigurator`. This class must have the responsibility of allowing the configuration of a game to become multiplayer. In particular, it must allow the user to enter the names of the players who must participate in the game, assigning each of them a progressive ID. This class must declare at least one `MultiPlayerGameImpl` type object as an instance variable, to which players must be added.

Then launching the following class:

```
public class MultiPlayerGameConfiguratorTest {
    public static void main(String args[]) {
        new MultiPlayerGameConfigurator();
    }
}
```

we want our program output to look like this:

```
Enter player name 1
Claudio
Enter player name 2 Or 'i' to start playing
Foo
Enter player name 3 Or 'i' to start playing
i
Players list:
Player 1: Claudio
Player 2: Foo
```

Exercise 10.y)

Now after realizing implementations of the `MultiPlayer` and `Configurator` interfaces, we just have to combine everything and evolve our application defined in Exercise 10.v, into a multi-user and configurable game. So first, let's make the `Player` class abstract by adding the following `play()` abstract method to it:

```
public abstract boolean play();
```

on the other hand, a player who does not define a `play()` method does not make much sense!

Then define the implementation of the `Player` class for our game, which we will call `GuessNumberPlayer`. It must be responsible for:

1. count the number of attempts made to guess the number.

2. Implement the `play()` method along the lines of the `GuessNumberGame` class `play()` method. In fact, now the player will play, and therefore must implement the code that allows him to play. The `play()` method must return `true`, only when the player guesses the number.

3. Furthermore, the `GuessNumberGame` class must be redefined, so that it implements the `Configurator` interface, and manages the game in a multi-player way using objects of the `GuessNumberPlayer` type. In particular it must:

4. redefine the `config()` method to manage the application configuration.

5. Redefine the `play()` method to take advantage of the `play()` method of the objects of type `GuessNumberGame`.

Exercise 10.z)

In this last exercise, the only goal is to place each class in a package with a meaningful name, in order to make the architectural structure of our application evident. All the application packages are listed below:

1. `games.generic.exceptions`

2. `games.generic.data`

3. `games.guessnumber.data`

4. `games.guessnumber.business`

5. `games.guessnumber.util`

6. `games.generic.business`

7. `games.generic.business`

your job is therefore to assign the right package to the right type.

Chapter 10
Exercise Solutions

A Guided Example to Object-Oriented Programming

We could implement the Point class in the following way:

```java
public class Point {
    private int x;
    private int y;

    public Point (int x, int y) {
        setX(x);
        setY(y);
    }

    private void setY(int y) {
        this.y = y;
    }

    public int getY() {
        return y;
    }

    private void setX(int x) {
        this.x = x;
    }
```

```
    public int getX() {
        return x;
    }

    public String toString() {
        return "(" +x+ "," + y + ")";
    }
}
```

The Ruler class follows (note that we have chosen to declare the method as static, since it depends only on the input parameters and not on any instance variables):

```
import static java.lang.Math.*;

public class Ruler {
    public static double getDistance(Point p1, Point p2) {
        return sqrt(pow(p1.getX()-p2.getX(), 2) + pow(p1.getY()-p2.getY(), 2));
    }
}
```

Finally follows the Exercise10A class, which instantiates two points, calculates the distance and uses an assertion as requested:

```
public class Exercise10A {
private final static double CORRECT_RESULT = 2;
    public static void main(String args[]) {
        Point p1 = new Point(0,0);
        Point p2 = new Point(2,0);
        double distance = Ruler.getDistance(p1,p2);
        System.out.println("Distance between points: " + p1 + " e " + p2
            + " = " + distance);
        assert distance == CORRECT_RESULT : "Error! The result should be "
            + CORRECT_RESULT;
        System.out.println("Correct distance!");
    }
}
```

Solution 10.b)

The code of the Exercise10B class is as follows:

```
public class Exercise10B {
    public static void main(String args[]) {
        if (args.length != 4) {
            System.out.println("Insert 4 coordinates to calculate the distance"
                + " between two points");
        }
```

```
        try {
            Point p1 = new Point(Integer.parseInt(args[0]),
                Integer.parseInt(args[1]));
            Point p2 = new Point(Integer.parseInt(args[2]),
                Integer.parseInt(args[3]));
            System.out.println("Distance between two points: " + p1 + " and "
                + p2 + " = " + Ruler.getDistance(p1, p2));
        } catch (NumberFormatException exc) {
            System.out.println("All coordinates must be integers! Error: "
                + exc.getMessage());
        }
    }
}
```

By launching the application with the following command:

```
java Exercise 10B 2 1 1 2
```

we will get the output:

```
Distance between two points: (2,1) and (1,2) = 1.4142135623730951
```

Solution 10.c)

Following is the code of the Exercise10C class:

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class Exercise10C {
    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Enter the x coordinate of the first point");
            int x1 = scanner.nextInt();
            System.out.println("Enter the y coordinate of the first point");
            int y1 = scanner.nextInt();
            Point p1 = new Point(x1, y1);
            System.out.println("Point created:" + p1);
            System.out.println("Enter the x coordinate of the second point");
            int x2 = scanner.nextInt();
            System.out.println("Enter the y coordinate of the second point");
            int y2 = scanner.nextInt();
            Point p2 = new Point(x2, y2);
            System.out.println("Point created:" + p2);
            System.out.println("Distance between two points: " + p1 + " and "
                + p2 + " = " + Ruler.getDistance(p1,p2));
```

```
        } catch (InputMismatchException exc) {
            System.out.println("All coordinates must be integers!");
        }
    }
}
```

Solution 10.d)

Following is the code of the Exercise10D class, refactoring of the Exercise10C class:

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class Exercise10D {
    private Scanner scanner;

    public Exercise10D () {
        scanner = new Scanner(System.in);
    }

    public void start() {
        try {
            Point p1 = getPoint("1");
            Point p2 = getPoint("2");
            printDistance(p1, p2) ;
        }
        catch (InputMismatchException exc) {
            System.out.println("All coordinates must be integers!");
        }
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Enter the coordinate " + coordinateName
            + " of point " + pointName);
        return scanner.nextInt();
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }

    public void printDistance(Point p1, Point p2) {
        System.out.println("Distance between two points: " + p1 + " and " + p2
            + " = " + Ruler.getDistance(p1,p2));
    }
```

```
    public static void main(String args[]) {
        Exercise10D exercise10D = new Exercise10D();
        exercise10D.start();
    }
}
```

Solution 10.e)

A possible implementation of the Exercise10E class follows:

```
import java.util.Scanner;

public class Exercise10E {
    private Scanner scanner;

    public Exercise10E () {
        scanner = new Scanner(System.in);
    }

    public void start() {
        Point p1 = getPoint("1");
        Point p2 = getPoint("2");
        printDistance(p1, p2) ;
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Enter the coordinate " + coordinateName
            + " of point " + pointName);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                return scanner.nextInt();
            } else {
                System.out.println("Invalid coordinate " +  scanner.next()
                    + "! All coordinates must be integers! "
                    + "Please enter only integers");
            }
        }
        return -1;
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }
```

```
    public void printDistance(Point p1, Point p2) {
        System.out.println("Distance between two points: " + p1 + " and " + p2
            + " = " + Ruler.getDistance(p1,p2));
    }

    public static void main(String args[]) {
        Exercise10E exercise10E = new Exercise10E();
        exercise10E.start();
    }
}
```

Solution 10.f)

The code of our Segment class could be the following:

```
public class Segment {
    private Point extremePoint1;
    private Point extremePoint2;

    private double length;

    public Segment(Point extremePoint1, Point extremePoint2 ) {
        this.extremePoint1 = extremePoint1;
        this.extremePoint2 = extremePoint2;
        setLength();
    }

    public Point getExtremePoint1() {
        return extremePoint1;
    }

    public Point getExtremePoint2() {
        return extremePoint2;
    }

    private void setLength() {
        this.length = Ruler.getDistance(extremePoint1, extremePoint2);
    }

    public String toString() {
        return "Segment from P1" +extremePoint1+ " to P2" +extremePoint2
            + " with length = " + length;
    }
}
```

Finally, we implemented the Exercise10F class as follows:

```
import java.util.*;
```

```java
public class Exercise10F {
    private Scanner scanner;

    public Exercise10F () {
        scanner = new Scanner(System.in);
    }

    public void start() {
        System.out.println("Please define a segment specifying its extreme"
            + " points:");
        Point p1 = getPoint("1");
        Point p2 = getPoint("2");
        print(new Segment(p1, p2) );
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Edit coordinate " + coordinateName
            + " of the extreme point " + pointName);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                return scanner.nextInt();
            } else {
                System.out.println("Invalid coordinate " +  scanner.next()
                + "! All coordinates must be integers! "
                + "Please enter only integers");
            }
        }
        return -1;
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }

    public void print(Segment segment) {
        System.out.println(segment);
    }

    public static void main(String args[]) {
        Exercise10F exercise10F = new Exercise10F();
        exercise10F.start();
    }
}
```

Solution 10.g)

We have modified the Point class as follows (changes in bold):

```java
public class Point {
    private String name;
    private int x;
    private int y;

    public Point (String name, int x, int y) {
        setName(name);
        setX(x);
        setY(y);
    }

    public Point (int x, int y) {
        this("", x, y);
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    private void setY(int y) {
        this.y = y;
    }

    public int getY() {
        return y;
    }

    private void setX(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }

    public String toString() {
        return name + "(" +x+ "," + y + ")";
    }
}
```

Finally, we implemented the Exercise10G class as follows:

312

```java
import java.util.*;
import javax.swing.*;

public class Exercise10G {
    private Scanner scanner;
    private CartesianPlane cartesianPlane;

    public Exercise10G () {
        scanner = new Scanner(System.in);
    }

    public void start() {
        SwingUtilities.invokeLater(() -> cartesianPlane = new CartesianPlane());
        Point p1 = getPoint("P1");
        cartesianPlane.drawPoint(p1);
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Edit coordinate " + coordinateName
            + " of the extreme point " + pointName);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                return scanner.nextInt();
            } else {
                System.out.println("Invalid coordinate " +  scanner.next()
                + "! All coordinates must be integers! "
                + "Please enter only integers");
            }
        }
        return -1;
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }

    public static void main(String args[]) {
        Exercise10G exercise10G = new Exercise10G();
        exercise10G.start();
    }
}
```

313

Solution 10.h)

The code of the `Exercise10H` class is as follows:

```java
import java.util.*;
import javax.swing.*;

public class Exercise10H {
    private Scanner scanner;
    private CartesianPlane cartesianPlane;
    private static int counter = 1;

    public Exercise10H () {
        scanner = new Scanner(System.in);
    }

    public void start() {
        SwingUtilities.invokeLater(() -> cartesianPlane = new CartesianPlane());
        while(true) {
            Point p1 = getPoint("P" +(counter++));
            cartesianPlane.drawPoint(p1);
        }
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Edit coordinate " + coordinateName
            + " of the extreme point " + pointName);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                return scanner.nextInt();
            } else {
                System.out.println("Invalid coordinate " +  scanner.next()
                + "! All coordinates must be integers! "
                + "Please enter only integers");
            }
        }
        return -1;
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }

    public static void main(String args[]) {
        Exercise10H exercise10H = new Exercise10H();
```

```
            exercise10H.start();
    }
}
```

Follows the `Exercise10I:` class:

```java
import java.util.*;
import javax.swing.*;

public class Exercise10I {
    private Scanner scanner;
    private CartesianPlane cartesianPlane;
    private static int counter = 1;

    public Exercise10I () {
        scanner = new Scanner(System.in);
    }

    public void start() {
        SwingUtilities.invokeLater(() -> cartesianPlane = new CartesianPlane());
        while(true) {
            System.out.println("Define a segment");
            Point p1 = getPoint("P" +(counter++));
            Point p2 = getPoint("P" +(counter++));
            Segment s1 = new Segment(p1, p2);
            cartesianPlane.drawSegment(s1);
        }
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Edit coordinate " + coordinateName
            + " of the extreme point " + pointName);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                return scanner.nextInt();
            } else {
                System.out.println("Invalid coordinate " +  scanner.next()
                + "! All coordinates must be integers! Please enter only"
                +  " integers");
            }
        }
        return -1;
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
```

```
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }

    public static void main(String args[]) {
        Exercise10I exercise10I = new Exercise10I();
        exercise10I.start();
    }
}
```

Solution 10.j)

We have implemented the `Polyline` class in the following way:

```
import java.util.*;

public class Polyline {
    private List<Segment> segmentList;

    public Polyline (Segment... segments) throws InvalidPolylineException {
        this.segmentList = new ArrayList(Arrays.asList(segments));
        if (segments.length < 2 || !checkConsecutiveSegments()) {
            throw new InvalidPolylineException(segmentList);
        }
    }

    public void addSegment(Segment newSegment) throws InvalidPolylineException {
        Segment lastSegment = segmentList.get(segmentList.size()-1);
        Point lastSegmentExtremePoint2 = lastSegment.getExtremePoint2();
        Point newSegmentExtremePoint1 = newSegment.getExtremePoint1();
            segmentList.add(newSegment);
        if (!equal(newSegmentExtremePoint1, lastSegmentExtremePoint2)) {
            throw new InvalidPolylineException(segmentList);
        }
    }

    private boolean checkConsecutiveSegments() {
        int segmentListSize = segmentList.size();
        for (int i = 0; i < segmentListSize-1; ) {
            Point previousSegmentExtremePoint2 =
                segmentList.get(i++).getExtremePoint2();
            Point nextSegmentExtremePoint1 =
                segmentList.get(i).getExtremePoint1();
            if (!equal(previousSegmentExtremePoint2,nextSegmentExtremePoint1)) {
                return false;
            }
```

316

```
        }
        return true;
    }

    private boolean equal(Point p1, Point p2) {
        return p1.getX()==p2.getX() && p1.getY()== p2.getY();
    }

    public String toString() {
        String descrizione ="Polyline defined by:\n";
        for (Segment segment : segmentList) {
            descrizione += segment + "\n";
        }
        return descrizione;
    }
}
```

Where the `InvalidPolylineException` class is implemented as follows:

```
import java.util.List;

public class InvalidPolylineException extends Exception {
    public InvalidPolylineException (List<Segment> segments) {
        super(segments.size() < 2 ?
            "A polyline must consist of at least 2 segments"
            : "These segments " + segments + " do not constitute a polyline");
    }

    public String toString() {
        return "Invalid polyline:\n" + getMessage();
    }
}
```

Finally, the implementation of the `PolylineTest` class is not simple:

```
public class PolylineTest {
    private static final String TEST_OK ="TEST OK:\n";
    private static final String TEST_KO ="TEST FAILED: ";

    public static void main(String args[]) {
        testCorrectPolyline();
        testCorrectPolylineWithFourPoints();
        testPolylineWithoutSegments();
        testPolylineWithOneSegment();
        testPolylineWithTwoNonConsecutiveSegmentsInConstructor();
        testPolylineWithThreeNonConsecutiveSegmentsInConstructor();
        testCorrectPolylineAddingAConsecutiveSegment();
        testInvalidPolylineAddingANonConsecutiveSegment();
    }
```

```java
    public static void testCorrectPolyline() {
        try {
            System.out.println("testCorrectPolyline:");
            Point p1 = new Point(0,0);
            Point p2 = new Point(1,1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1,2);
            Segment s2 = new Segment(p2, p3);
            Polyline pol1 = new Polyline(s1, s2);
            System.out.println(TEST_OK + pol1);
        } catch (Exception exc) {
            assert false : TEST_KO +  exc.getMessage();
        }
    }

    public static void testCorrectPolylineWithFourPoints() {
        try {
            System.out.println("testCorrectPolylineWithFourPoints:");
            Point p1 = new Point(0,0);
            Point p2 = new Point(1,1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1,1);
            Point p4 = new Point(2,3);
            Segment s2 = new Segment(p3, p4);
            Polyline pol1 = new Polyline(s1, s2);
            System.out.println(TEST_OK + pol1);
        } catch (Exception exc) {
            assert false : TEST_KO +  exc.getMessage();
        }
    }

    public static void testPolylineWithoutSegments() {
        try {
            System.out.println("testPolylineWithoutSegments:");
            Polyline pol1 = new Polyline();
            System.out.println(TEST_KO + pol1);
            assert false :"A polyline without segments has been created!";
        } catch (Exception exc) {
            System.out.println(TEST_OK +  exc.getMessage());
            System.out.println();
        }
    }

    public static void testPolylineWithOneSegment() {
        try {
            System.out.println("testPolylineWithOneSegment:");
            Point p1 = new Point(0,0);
            Point p2 = new Point(1,1);
            Segment s1 = new Segment(p1, p2);
```

```
            Polyline pol1 = new Polyline(s1);
            System.out.println(TEST_KO + pol1);
            assert false :"A polyline with a single segment has been created!";
        } catch (Exception exc) {
            System.out.println(TEST_OK +  exc.getMessage());
            System.out.println();
        }
    }

    public static void testPolylineWithTwoNonConsecutiveSegmentsInConstructor() {
        try {
            System.out.println(
                "testPolylineWithTwoNonConsecutiveSegmentsInConstructor:");
            Point p1 = new Point(0,0);
            Point p2 = new Point(1,1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1,2);
            Point p4 = new Point(2,2);
            Segment s2 = new Segment(p3, p4);
            Polyline pol1 = new Polyline(s1, s2);
            System.out.println(TEST_KO+ pol1);
            assert false :"A polyline with two non-consecutive segments "
                + "has been created (in the constructor)!";
        } catch (Exception exc) {
            System.out.println(TEST_OK +  exc.getMessage());
            System.out.println();
        }
    }

    public static void testPolylineWithThreeNonConsecutiveSegmentsInConstructor() {
        try {
            System.out.println(
                "testPolylineWithThreeNonConsecutiveSegmentsInConstructor:");
            Point p1 = new Point(0,0);
            Point p2 = new Point(1,1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1,1);
            Point p4 = new Point(2,2);
            Segment s2 = new Segment(p3, p4);
            Point p5 = new Point(2,3);
            Point p6 = new Point(3,2);
            Segment s3 = new Segment(p5, p6);
            Polyline pol1 = new Polyline(s1, s2, s3);
            System.out.println(TEST_KO+ pol1);
            assert false :"A polyline with three non-consecutive segments"
                + " has been created (in the constructor)!";
        } catch (Exception exc) {
            System.out.println(TEST_OK +  exc.getMessage());
            System.out.println();
```

```
        }
    }

    public static  void testCorrectPolylineAddingAConsecutiveSegment() {
        try {
            System.out.println("testCorrectPolylineAddingAConsecutiveSegment:");
            Point p1 = new Point(0,0);
            Point p2 = new Point(1,1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1,2);
            Segment s2 = new Segment(p2, p3);
            Polyline pol1 = new Polyline(s1, s2);
            Point p4 = new Point(3,2);
            Segment s3 = new Segment(p3, p4);
            pol1.addSegment(s3);
            System.out.println(TEST_OK+ pol1);
        } catch (Exception exc) {
            assert false : TEST_KO +  exc.getMessage();
        }
    }

    public static void testInvalidPolylineAddingANonConsecutiveSegment() {
        try {
            System.out.println(
                "testInvalidPolylineAddingANonConsecutiveSegment:");
            Point p1 = new Point(0,0);
            Point p2 = new Point(1,1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1,2);
            Segment s2 = new Segment(p2, p3);
            Polyline pol1 = new Polyline(s1, s2);
            Point p4 = new Point(3,2);
            Point p5 = new Point(3,3);
            Segment s3 = new Segment(p4, p5);
            pol1.addSegment(s3);
            System.out.println(TEST_KO+ pol1);
            assert false :
                "A polyline with a non-consecutive segment has been created!";
        } catch (Exception exc) {
            System.out.println(TEST_OK +  exc.getMessage());
        }
    }
}
```

Solution 10.k)

Actually, the errors highlighted by our test were mostly due to errors in the test itself! This is to underline that we can create bugs even in tests! So, you have to be very careful. In

320

fact in the `testDistanceWithNull()` it is correct that a `NullPointerException` is thrown (although it would be the case to create a custom exception to handle this problem), while in the `testDistanceFromAThreeDimensionalPoint()` method, we had expected that the result of the distance would have been –1, but obviously the value of a distance cannot coincide with a negative number! In this case we solved it by changing the sign, since we had decided to allow this type of operation (we could also have implemented a solution that would throw another custom exception since we could have assumed that it is not possible to calculate the distance between a point that is on a two-coordinate reference, from a point that is on a three-coordinate reference). In any case, we report below the new `PointTest` class, with the changes in bold.

```java
import org.junit.Assert;
import org.junit.Test;

public class PointTest {
    @Test
    public void testDistanceOnX() {
        Point p1 = new Point(1,1);
        Point p2 = new Point(1,2);
        double distance = p1.distance(p2);
        Assert.assertTrue(distance == 1);
    }
    @Test(expected = NullPointerException.class)
    public void testDistanceWithNull() {
        Point p1 = new Point(1,1);
        Point p2 = null;
        double distance = p1.distance(p2);
//        Assert.assertTrue(distance == -1);
    }
    @Test
    public void testDistanceFromTheSamePoint() {
        Point p1 = new Point(1,1);
        Point p2 = new Point(1,1);
        double distance = p1.distance(p2);
        Assert.assertTrue(distance == 0);
    }
    @Test
    public void testDistanceFromAThreeDimensionalPoint() {
        Point p1 = new Point(1,1);
        Point p2 = new ThreeDimensionalPoint(1,2,2);
        double distance = p1.distance(p2);
        Assert.assertTrue(distance == 1);
    }
}
```

Solution 10.l)

The implementation of the new `PolylineTest` class follows (among the book code file, you will find the entire project to be imported into Eclipse):

```java
import org.junit.Assert;
import org.junit.Test;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class PolylineTest {
    private static final String TEST_OK = "TEST OK:\n";
    private static final String TEST_KO = "TEST FALLITO: ";

//    public static void main(String args[]) {
//        testCorrectPolyline();
//        testCorrectPolylineWithFourPoints();
//        testPolylineWithoutSegments();
//        testPolylineWithOneSegment();
//        testPolylineWithTwoNonConsecutiveSegmentsInConstructor();
//        testPolylineWithThreeNonConsecutiveSegmentsInConstructor();
//        testCorrectPolylineAddingAConsecutiveSegment();
//        testInvalidPolylineAddingANonConsecutiveSegment();
//    }

    @Test
    public void testCorrectPolyline() {
        try {
            System.out.println("testCorrectPolyline:");
            Point p1 = new Point(0, 0);
            Point p2 = new Point(1, 1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1, 2);
            Segment s2 = new Segment(p2, p3);
            Polyline pol1 = new Polyline(s1, s2);
            Assert.assertNotNull(pol1);
        } catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    @Test
    public void testCorrectPolylineWithFourPoints() {
        try {
            System.out.println("testCorrectPolylineWithFourPoints:");
            Point p1 = new Point(0, 0);
            Point p2 = new Point(1, 1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1, 1);
            Point p4 = new Point(2, 3);
```

```
            Segment s2 = new Segment(p3, p4);
            Polyline pol1 = new Polyline(s1, s2);
            Assert.assertNotNull(pol1);
        } catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    @Test
    public void testPolylineWithoutSegments() {
        try {
            System.out.println("testPolylineWithoutSegments:");
            Polyline pol1 = new Polyline();
            assert false : "A polyline without segments has been"
                        + "  created!";
        } catch (Exception exc) {
            Assert.assertNotNull(exc);
            System.out.println(TEST_OK + exc.getMessage());
            System.out.println();
        }
    }

    @Test
    public void testPolylineWithOneSegment() {
        try {
            System.out.println("testPolylineWithOneSegment:");
            Point p1 = new Point(0, 0);
            Point p2 = new Point(1, 1);
            Segment s1 = new Segment(p1, p2);
            Polyline pol1 = new Polyline(s1);
            System.out.println(TEST_KO + pol1);
            assert false : "A polyline with a single segment has"
                        + " been created!";
        } catch (Exception exc) {
            Assert.assertNotNull(exc);
            System.out.println(TEST_OK + exc.getMessage());
            System.out.println();
        }
    }

    public void testPolylineWithTwoNonConsecutiveSegmentsInConstructor() {
        try {
            System.out.println(
                "testPolylineWithTwoNonConsecutiveSegmentsInConstructor:");
            Point p1 = new Point(0, 0);
            Point p2 = new Point(1, 1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1, 2);
            Point p4 = new Point(2, 2);
```

Java for Aliens - Claudio De Sio Cesari (www.javaforaliens.com)

```java
            Segment s2 = new Segment(p3, p4);
            Polyline pol1 = new Polyline(s1, s2);
            System.out.println(TEST_KO + pol1);
            assert false : "A polyline with two non-consecutive"
                    + " segments has been created (in the constructor)";
        } catch (Exception exc) {
            Assert.assertNotNull(exc);
            System.out.println(TEST_OK + exc.getMessage());
            System.out.println();
        }
    }

    public void testPolylineWithThreeNonConsecutiveSegmentsInConstructor() {
        try {
            System.out.println(
                "testPolylineWithThreeNonConsecutiveSegmentsInConstructor:");
            Point p1 = new Point(0, 0);
            Point p2 = new Point(1, 1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1, 1);
            Point p4 = new Point(2, 2);
            Segment s2 = new Segment(p3, p4);
            Point p5 = new Point(2, 3);
            Point p6 = new Point(3, 2);
            Segment s3 = new Segment(p5, p6);
            Polyline pol1 = new Polyline(s1, s2, s3);
            System.out.println(TEST_KO + pol1);
            assert false : "A polyline with three non-consecutive"
                    + " segments has been created (in the constructor)!";
        } catch (Exception exc) {
            Assert.assertNotNull(exc);
            System.out.println(TEST_OK + exc.getMessage());
            System.out.println();
        }
    }

    @Test
    public void testCorrectPolylineAddingAConsecutiveSegment() {
        try {
            System.out.println(
                    "testCorrectPolylineAddingAConsecutiveSegment:");
            Point p1 = new Point(0, 0);
            Point p2 = new Point(1, 1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1, 2);
            Segment s2 = new Segment(p2, p3);
            Polyline pol1 = new Polyline(s1, s2);
            Point p4 = new Point(3, 2);
            Segment s3 = new Segment(p3, p4);
```

324

```
                pol1.addSegment(s3);
                System.out.println(TEST_OK + pol1);
            } catch (Exception exc) {
                exc.printStackTrace();
            }
        }
    }

    @Test
    public void testInvalidPolylineAddingANonConsecutiveSegment() {
        try {
            System.out.println(
                        "testInvalidPolylineAddingANonConsecutiveSegment:");
            Point p1 = new Point(0, 0);
            Point p2 = new Point(1, 1);
            Segment s1 = new Segment(p1, p2);
            Point p3 = new Point(1, 2);
            Segment s2 = new Segment(p2, p3);
            Polyline pol1 = new Polyline(s1, s2);
            Point p4 = new Point(3, 2);
            Point p5 = new Point(3, 3);
            Segment s3 = new Segment(p4, p5);
            pol1.addSegment(s3);
            System.out.println(TEST_KO + pol1);
            assert false : "A polyline with a non-consecutive segment "
                            + "has been created!";
        } catch (Exception exc) {
            Assert.assertNotNull(exc);
            System.out.println(TEST_OK + exc.getMessage());
        }
    }
}
```

The implementation of the `Exercise10M` class follows (changes compared to the `Exercise10I` class in bold):

```
import java.util.*;
import javax.swing.*;

public class Exercise10M {
    private Scanner scanner;
    private CartesianPlane cartesianPlane;
    private static int counter = 1;

    public Exercise10M () {
        scanner = new Scanner(System.in);
    }
```

```java
    public void start() {
        SwingUtilities.invokeLater(() -> cartesianPlane = new CartesianPlane());
        System.out.println("Let's define the first two segments of a "
            + " polyline:");
        System.out.println("Define the first extreme point of the first"
            + "  segment of the polyline:");
        Point p1 = getPoint("P" +(counter++));
        System.out.println("Define the second extreme point of the first"
            + "  segment of the polyline:");
        Point p2 = getPoint("P" +(counter++));
        Segment s1 = new Segment(p1, p2);
        cartesianPlane.drawSegment(s1);
        System.out.println("Define the second extreme point of the"
            + "  second segment of the polyline:");
        Point p3 = getPoint("P" +(counter++));
        Segment s2 = new Segment(p2, p3);
        cartesianPlane.drawSegment(s2);
        Point lastExtreme = p3;
        while(true) {
            Point nextExtreme = getPoint("P" +(counter++));
            Segment nextSegment = new Segment(lastExtreme, nextExtreme);
            cartesianPlane.drawSegment(nextSegment);
            lastExtreme = nextExtreme;
        }
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Edit coordinate " + coordinateName
            + " of the extreme point " + pointName);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                return scanner.nextInt();
            } else {
                System.out.println("Invalid coordinate " +  scanner.next()
                + "! All coordinates must be integers! Please enter only"
                + "  integers");
            }
        }
        return -1;
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }
```

326

```
    public static void main(String args[]) {
        Exercise10M exercise10M = new Exercise10M();
        exercise10M.start();
    }
}
```

Solution 10.n)

Let's first create a simple exception for the incorrect creation of a segment in this way:

```
public class InvalidSegmentException extends Exception {

    public String toString() {
        return "The extreme points of a segment cannot coincide!";
    }
}
```

Then let's throw this exception in the Segment class in case there is a coincidence of the two extreme point (changes in bold):

```
public class Segment {
    private Point extremePoint1;
    private Point extremePoint2;

    private double length;

    public Segment(Point extremePoint1, Point extremePoint2 )
            throws InvalidSegmentException {
        this.extremePoint1 = extremePoint1;
        this.extremePoint2 = extremePoint2;
        setLength();
    }

    public Point getExtremePoint1() {
        return extremePoint1;
    }

    public Point getExtremePoint2() {
        return extremePoint2;
    }

    private void setLength() throws InvalidSegmentException {
        this.length = Ruler.getDistance(extremePoint1, extremePoint2);
        if (length == 0) {
            throw new InvalidSegmentException();
        }
    }
}
```

```
    public String toString() {
        return "Segment from P1" +extremePoint1+ " to P2" + extremePoint2
            + " with length = " + length;
    }
}
```

Finally, let's make our Exercise10M class evolve into the Exercise10N class, creating a recursive method createSegment(), which manages the exception by calling itself. We also note that we have transformed the temporary variable lastExtreme, as an instance variable (changes in bold):

```
import java.util.*;
import javax.swing.*;

public class Exercise10N {
    private Scanner scanner;
    private CartesianPlane cartesianPlane;
    private Point lastExtreme;
    private static int counter = 1;

    public Exercise10N () {
        scanner = new Scanner(System.in);
    }

    public void start() {
        SwingUtilities.invokeLater(() -> cartesianPlane = new CartesianPlane());
        System.out.println("Let's define the first two segments of a"
            + " polyline:");
        System.out.println("Define the first extreme point of the first"
            + "  segment of the polyline:");
        Point p1 = getPoint("P" +(counter++));
        System.out.println("Define the second extreme point of the first"
            + "  segment of the polyline:");
        createSegment(p1);
        /* Point p2 = getPoint("P" +(counter++));
        Segment s1 = new Segment(p1, p2);
        cartesianPlane.drawSegment(s1);*/
        System.out.println("Define the second extreme point of the second"
            + "  segment of the polyline:");
        createSegment(lastExtreme);
        /* Point p3 = getPoint("P" +(counter++));
        Segment s2 = new Segment(p2, p3);
        cartesianPlane.drawSegment(s2);
        Point lastExtreme = p3;*/
        while(true) {
            /* Point nextExtreme = getPoint("P" +(counter++));
            Segment nextSegment = new Segment(lastExtreme, nextExtreme);
            cartesianPlane.drawSegment(nextSegment);
```

```
            lastExtreme = nextExtreme; */
            createSegment(lastExtreme);
        }
    }

    public void createSegment(Point firstExtreme) {
        try {
            lastExtreme = getPoint("P" +(counter++));
            Segment segment = new Segment(firstExtreme, lastExtreme);
            cartesianPlane.drawSegment(segment);
        } catch (InvalidSegmentException exc) {
            System.out.println(exc);
            counter--;
            createSegment(firstExtreme);
        }
    }

    private int getCoordinate(String coordinateName, String pointName) {
        System.out.println("Edit coordinate " + coordinateName
            + " of the extreme point " + pointName);
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                return scanner.nextInt();
            } else {
                System.out.println("Invalid coordinate " +  scanner.next()
                + "! All coordinates must be integers! "
                + " Please enter only integers");
            }
        }
        return -1;
    }

    private Point getPoint(String pointName) {
        int x = getCoordinate("x", pointName);
        int y = getCoordinate("y", pointName);
        Point p = new Point(x,y);
        System.out.println("Point created: " + p);
        return p;
    }

    public static void main(String args[]) {
        Exercise10N exercise10N = new Exercise10N();
        exercise10N.start();
    }
}
```

Solution 10.o)

The solution could be implemented as follows:

```java
import java.util.*;

public class Exercise10O {
    public static void main(String[] args) {
        Exercise10O exercise10O = new Exercise10O();
        exercise10O.start();
    }

    public void start() {
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.println("Enter a Kelvin value");
            float kelvin = scanner.nextFloat();
            float celsius = kelvin - 273.15F;
            float fahrenheit = celsius * 9/5 + 32;
            System.out.println(kelvin + "  Kelvin equals:\n"
                + celsius  + " Celsius\n"
                + fahrenheit + " Fahrenheit");
        } catch (InputMismatchException exc) {
            System.out.println("The value must be numeric");
            start();
        }
    }
}
```

Solution 10.p)

The solution could be implemented as follows:

```java
import java.util.*;
import java.math.*;

public class Exercise10P {
    public static void main(String[] args) {
        Exercise10P exercise10P = new Exercise10P();
        exercise10P.start();
    }

    public void start() {
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.println("Enter a Kelvin value");
            BigDecimal kelvin = scanner.nextBigDecimal();
            BigDecimal celsius = kelvin.subtract(BigDecimal.valueOf(273.15));
```

```
            BigDecimal fahrenheit = celsius.multiply(BigDecimal.valueOf(9))
                  .divide(BigDecimal.valueOf(5)).add(BigDecimal.valueOf(32));
            System.out.println(kelvin + " Kelvin equals:\n"
            + celsius  + " Celsius\n" + fahrenheit + " Fahrenheit");
        } catch (InputMismatchException exc) {
            System.out.println("The value must be numeric");
            start();
        }
    }
}
```

Solution 10.q)

The solution could be implemented as follows:

```
import java.util.*;
import java.math.*;

public class Exercise10Q {
    public static void main(String[] args) {
        Exercise10Q exercise10Q = new Exercise10Q();
        exercise10Q.start();
    }

    public void start() {
        try {
            System.out.println("Enter a value in Kelvin (for example 12K)," +
                " Celsius (25C) or Fahrenheit (451F)");
            Scanner scanner = new Scanner(System.in);
            String input = scanner.next();
            int lastCharIndex = input.length()-1;
            BigDecimal value = new BigDecimal(
                input.substring(0, lastCharIndex));
            String scale = input.substring(lastCharIndex);
            switch(scale) {
                case"k":
                case"K":
                {
                    BigDecimal celsius = value.subtract(
                      BigDecimal.valueOf(273.15));
                    BigDecimal fahrenheit = celsius.multiply(
                      BigDecimal.valueOf(9)).divide(BigDecimal.valueOf(5), 2,
                      RoundingMode.HALF_UP).add(BigDecimal.valueOf(32));
                    System.out.println(value + " Kelvin equals:\n"
                      + celsius  + " Celsius\n" + fahrenheit + " Fahrenheit");
                    break;
                }
```

```
                    case"c":
                    case"C":
                    {
                        BigDecimal kelvin = value.add(BigDecimal.valueOf(273.15));
                        BigDecimal fahrenheit = value.multiply(
                            BigDecimal.valueOf(9)).divide(BigDecimal.valueOf(5),2,
                            RoundingMode.HALF_UP).add(BigDecimal.valueOf(32));
                        System.out.println(value + " Celsius equals:\n"
                            + kelvin  + " Kelvin\n" + fahrenheit + " Fahrenheit");
                        break;
                    }
                    case"f":
                    case"F":
                    {
                        BigDecimal celsius = value.subtract(BigDecimal.valueOf(32))
                            .multiply(BigDecimal.valueOf(5)).divide(
                            BigDecimal.valueOf(9), 2, RoundingMode.HALF_UP);
                        BigDecimal kelvin = celsius.add(BigDecimal.valueOf(273.15));
                        System.out.println(value + " Fahrenheit equals:\n"
                             + celsius  + " Celsius\n" + kelvin + " Kelvin");
                        break;
                    }
                    default:
                        System.out.println("Invalid scale "  + scale
                          + ". Use K for Kelvin, C for Celsius, F for Fahrenheit");
                        start();
                }
        } catch (NumberFormatException exc) {
            System.out.println("Wrong format!");
            start();
        }
    }
}
```

Solution 10.r)

The Converter class could be implemented as follows:

```
import java.math.*;

public class Converter {
    public static BigDecimal convertKelvinToCelsius(BigDecimal kelvin) {
        return kelvin.subtract(BigDecimal.valueOf(273.15));
    }

    public static BigDecimal convertKelvinToFahrenheit(BigDecimal kelvin) {
        BigDecimal celsius = convertKelvinToCelsius(kelvin);
        return celsius.multiply(BigDecimal.valueOf(9)).divide(
```

332

```
                BigDecimal.valueOf(5), 2, RoundingMode.HALF_UP).add(
                BigDecimal.valueOf(32));
    }

    public static BigDecimal convertCelsiusToKelvin(BigDecimal celsius) {
        return celsius.add(BigDecimal.valueOf(273.15));
    }

    public static BigDecimal convertCelsiusToFahrenheit(BigDecimal celsius) {
        return celsius.multiply(BigDecimal.valueOf(9)).divide(
            BigDecimal.valueOf(5),2, RoundingMode.HALF_UP).add(
            BigDecimal.valueOf(32));
    }

    public static BigDecimal convertFahrenheitToKelvin(BigDecimal fahrenheit) {
        BigDecimal celsius = convertFahrenheitToCelsius(fahrenheit);
        return celsius.add(BigDecimal.valueOf(273.15));
    }

    public static BigDecimal convertFahrenheitToCelsius(BigDecimal fahrenheit) {
        return fahrenheit.subtract(BigDecimal.valueOf(32))
            .multiply(BigDecimal.valueOf(5)).divide(BigDecimal.valueOf(9), 2,
            RoundingMode.HALF_UP);
    }
}
```

So, the `Exercise10R` class could be implemented in the following way:

```
import java.util.*;
import java.math.*;

public class Exercise10R {
    public static void main(String[] args) {
        Exercise10R exercise10R = new Exercise10R();
        exercise10R.start();
    }

    public void start() {
        try {
            System.out.println("Enter a value in Kelvin (for example 12K),"
                + " Celsius (25C) or Fahrenheit (451F)");
            Scanner scanner = new Scanner(System.in);
            String input = scanner.next();
            int lastCharIndex = input.length()-1;
            BigDecimal value = new BigDecimal(input.substring(0,
                lastCharIndex));
            String scale = input.substring(lastCharIndex);
            switch(scale) {
```

```
                case"k":
                case"K": {
                    BigDecimal celsius =
                        Converter.convertKelvinToCelsius(value);
                    BigDecimal fahrenheit =
                        Converter.convertKelvinToFahrenheit(value);
                    System.out.println(value + " Kelvin equals:\n"
                      + celsius  + " Celsius\n" + fahrenheit + " Fahrenheit");
                    break;
                }
                case"c":
                case"C": {
                    BigDecimal kelvin = Converter.convertCelsiusToKelvin(value);
                    BigDecimal fahrenheit =
                        Converter.convertCelsiusToFahrenheit(value);
                    System.out.println(value + " Celsius equals:\n"
                        + kelvin  + " Kelvin\n" + fahrenheit + " Fahrenheit");
                    break;
                }
                case"f":
                case"F": {
                    BigDecimal celsius =
                        Converter.convertFahrenheitToCelsius(value);
                    BigDecimal kelvin =
                        Converter.convertFahrenheitToKelvin(value);
                    System.out.println(value + " Fahrenheit equals:\n"
                        + celsius  + " Celsius\n" + kelvin + " Kelvin");
                    break;
                }
                default:
                    System.out.println("Invalid scale "  + scale
                      + ". Use K for Kelvin, C for Celsius, F for Fahrenheit");
                    start();
            }
        }
        catch (NumberFormatException exc) {
            System.out.println("Wrong format!");
            start();
        }
    }
}
```

Solution 10.s)

The Printer class could be implemented in the following way:

```
import java.math.*;

public class Printer {
```

334

```
    public static void printInstructions() {
        System.out.println("Enter a value in Kelvin (for example 12K), " +
            "Celsius (25C) or Fahrenheit (451F)");
    }

    public static void printMessage(BigDecimal value1, String scale1,
      BigDecimal value2, String scale2,BigDecimal value3, String scale3) {
        System.out.println(value1 + " " + scale1+ " equals:\n"
            + value2 + " " + scale2 + "\n" + value3 + " " + scale3);
    }

    public static void printInvalidScale(String scale) {
        System.out.println("Invalid scale " + scale
            + ". Use K for Kelvin, C for Celsius, F for Fahrenheit");
    }

    public static void printWrongFormat() {
        System.out.println("Wrong format!");
    }
}
```

So, the Exercise10S class could be implemented in the following way:

```
import java.util.*;
import java.math.*;

public class Exercise10S {
    private static final String K ="Kelvin";
    private static final String C ="Celsius";
    private static final String F ="Fahrenheit";

    public static void main(String[] args) {
        Exercise10S exercise10S = new Exercise10S();
        exercise10S.start();
    }

    public void start() {
        try {
            Printer.printInstructions();
            Scanner scanner = new Scanner(System.in);
            String input = scanner.next();
            int lastCharIndex = input.length()-1;
            BigDecimal value = new BigDecimal(
                input.substring(0, lastCharIndex));
            String scale = input.substring(lastCharIndex);
            switch(scale) {
                case"k":
                case"K": {
                    BigDecimal celsius =
```

335

```
                        Converter.convertKelvinToCelsius(value);
                    BigDecimal fahrenheit =
                        Converter.convertKelvinToFahrenheit(value);
                    Printer.printMessage(value, K, celsius, C, fahrenheit, F);
                    break;
                }
                case"c":
                case"C": {
                    BigDecimal kelvin = Converter.convertCelsiusToKelvin(value);
                    BigDecimal fahrenheit =
                        Converter.convertCelsiusToFahrenheit(value);
                    Printer.printMessage(value, C, kelvin, K, fahrenheit, F);
                    break;
                }
                case"f":
                case"F": {
                    BigDecimal celsius =
                        Converter.convertFahrenheitToCelsius(value);
                    BigDecimal kelvin =
                        Converter.convertFahrenheitToKelvin(value);
                    Printer.printMessage(value, F, celsius, C, kelvin, K);
                    break;
                }
                default:
                Printer.printInvalidScale(scale);
                start();
            }
        } catch (NumberFormatException exc) {
            Printer.printWrongFormat();
            start();
        }
    }
}
```

Note that the `printMessage()` method undoubtedly has too many parameters, and its use appears to be quite complex. However, the solution seems better than that of the previous exercise.

Solution 10.t)

The following could be a possible implementation of the `Exercise10T` class:

```
import java.util.*;
import java.io.*;
import java.math.*;
import java.util.logging.*;

public class Exercise10T {
```

```
    private static final LogManager logManager = LogManager.getLogManager();
    private static final Logger LOGGER = Logger.getLogger("Exercise10T");
    private static final String K ="Kelvin";
    private static final String C ="Celsius";
    private static final String F ="Fahrenheit";

    public static void main(String[] args) {
        initLogging();
        LOGGER.info("Main() method called");
        Exercise10T exercise10T = new Exercise10T();
        exercise10T.start();
        LOGGER.info("Program terminated");
    }

    private static void initLogging() {
        try {
            logManager.readConfiguration(new FileInputStream(
                "logging.properties"));
        } catch (IOException exception) {
            LOGGER.log(Level.SEVERE,
                "Problem reading configuration file",exception);
        }
    }

    public void start() {
        try {
            LOGGER.info("Start() method called");
            Printer.printInstructions();
            Scanner scanner = new Scanner(System.in);
            String input = scanner.next();
            int lastCharIndex = input.length()-1;
            BigDecimal value = new BigDecimal(input.substring(0,
                lastCharIndex));
            String scale = input.substring(lastCharIndex);
            switch(scale) {
                case"k":
                case"K": {
                    LOGGER.fine("Kelvin scale in use");
                    BigDecimal celsius =
                        Converter.convertKelvinToCelsius(value);
                    BigDecimal fahrenheit =
                        Converter.convertKelvinToFahrenheit(value);
                    Printer.printMessage(value, K, celsius, C, fahrenheit, F);
                    break;
                }
                case"c":
                case"C": {
                    LOGGER.fine("Celsius scale in use");
                    BigDecimal kelvin = Converter.convertCelsiusToKelvin(value);
```

```
                        BigDecimal fahrenheit =
                            Converter.convertCelsiusToFahrenheit(value);
                        Printer.printMessage(value, C, kelvin, K, fahrenheit, F);
                        break;
                    }
                    case"f":
                    case"F": {
                        LOGGER.fine("Fahrenheit scale in use");
                        BigDecimal celsius =
                            Converter.convertFahrenheitToCelsius(value);
                        BigDecimal kelvin =
                            Converter.convertFahrenheitToKelvin(value);
                        Printer.printMessage(value, F, celsius, C, kelvin, K);
                        break;
                    }
                    default:
                    LOGGER.fine("Invalid scale");
                    Printer.printInvalidScale(scale);
                    start();
                }
            } catch (NumberFormatException exc) {
                LOGGER.severe(exc.getMessage());
                Printer.printWrongFormat();
                start();
            }
        }
    }
}
```

By launching this class with the default level ALL, in the log file we will find the following text:

```
apr 27, 2020 12:53:01 PM Exercise10T main
INFO: Main() method called
apr 27, 2020 12:53:01 PM Exercise10T start
INFO: Start() method called Metodo start() chiamato
apr 27, 2020 12:53:03 PM Exercise10T start
FINE: Kelvin scale in use
apr 27, 2020 12:53:03 PM Exercise10T main
INFO: Program terminated
```

But by changing the log level to INFO, and using the application correctly, we will find this text in the log file:

```
apr 27, 2020 12:54:01 PM Exercise10T main
INFO: Main() method called
apr 27, 2020 12:54:02 PM Exercise10T start
INFO: Start() method called
apr 27, 2020 12:54:05 PM Exercise10T main
INFO: Program terminated
```

That is, the FINE type logs have been ignored.

Finally, if we changed the log level to SEVERE we would only see the SEVERE level logs. So, the file would be unchanged if the application was used correctly, while it would contain the log messages defined to manage the exception if the user specifies an invalid scale. For example using the application in the following way:

```
Enter a value in Kelvin (for example 12K), Celsius (25C) or
 Fahrenheit (451F)
j
Wrong format!
Enter a value in Kelvin (for example 12K), Celsius (25C) or
 Fahrenheit (451F)
0k
0 Kelvin equals:
-273.15 Celsius
-459.67 Fahrenheit
```

we would get the following log file:

```
apr 27, 2020 12:56:21 PM Exercise10T start
SEVERE: Character j is neither a decimal digit number, decimal point, nor "e"
 notation exponential mark.
```

Solution 10.u)

We can implement the GuessNumber class in the following way:

```java
import java.util.Random;

public class GuessNumber {
    private static final Random RANDOM = new Random();

    public static int generateRandomNumber(int max) {
        return 1+ RANDOM.nextInt(max);
    }
}
```

While we can define the GuessNumberGame class as follows:

```java
import java.util.Scanner;
import java.util.Random;

public class GuessNumberGame implements Game {
    private static final int MAX_NUMBER = 100;
    private Scanner scanner;
    private int attemptsNumber;
    private int numberToGuess;
```

339

```java
    public GuessNumberGame () {
        init();
        start();
        play();
        end();
    }

    @Override
    public void init() {
        scanner = new Scanner(System.in);
        numberToGuess = GuessNumber.generateRandomNumber(MAX_NUMBER);
    }

    @Override
    public void start() {
        System.out.println("I'm thinking of a number between 1 and "
            + MAX_NUMBER + ", guess what!");
    }

    @Override
    public void play() {
        attemptsNumber++;
        int number = scanner.nextInt();
        if (number < numberToGuess) {
            System.out.println("Too low, try again");
        } else if (number > numberToGuess) {
            System.out.println("Too high, try again");
        } else {
            return;
        }
        play();
    }

    @Override
    public void end() {
        System.out.println("Yes! You guessed it after " + attemptsNumber
            + " attempts");
    }

    public static void main(String args[]) {
        new GuessNumberGame();
    }
}
```

Note that the `main()` method simply instantiated the `GuessNumberGame` object by calling its constructor. The latter invoked the methods defined by the interface which were implemented in a very simple and consistent way. Only the recursive `play()` method deserves attention, which calls itself up until the user has guessed the number.

340

Solution 10.v)

We can redefine the `GuessNumberGame` class as follows (changes in bold), removing the recursive nature of the `play()` method:

```java
import java.util.Scanner;
import java.util.Random;

public class GuessNumberGame implements Game {
    private static final int MAX_NUMBER = 100;
    private Scanner scanner;
    private int attemptsNumber;
    private int numberToGuess;

    public GuessNumberGame () {
        init();
        start();
        play();
        end();
    }

    @Override
    public void init() {
        scanner = new Scanner(System.in);
        numberToGuess = GuessNumber.generateRandomNumber(MAX_NUMBER);
    }

    @Override
    public void start() {
        System.out.println("I'm thinking of a number between 1 and "
          + MAX_NUMBER + ", guess what!");
    }

    @Override
    public void play() {
        while (scanner.hasNext()) {
            if (scanner.hasNextInt()) {
                attemptsNumber++;
                int number = scanner.nextInt();
                if (number < numberToGuess) {
                    System.out.println("Too low, try again");
                } else if (number > numberToGuess) {
                    System.out.println("Too high, try again");
                } else {
                    return;
                }
                //play()
            } else {
                System.out.println("Invalid input '" + scanner.next ()
```

```
                            + "'. You can only enter integer numbers, please try again");
                }
            }
        }

        @Override
        public void end() {
            System.out.println("Yes! You guessed it after " + attemptsNumber
                + " attempts");
        }

        public static void main(String args[]) {
            new GuessNumberGame();
        }
    }
}
```

Solution 10.w)

We can implement the Player class in the following way:

```java
public class Player {
    private int id;
    private String name;

    public Player (int id, String name) {
        this.setId(id);
        this.setName(name);
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Player " + id + ": " + name;
    }
}
```

Then we implement the `PlayerException` class:

```
public class PlayerException extends Exception {
    public PlayerException (String message){
        super("Player exception: "  + message);
    }
}
```

The implementation of the `MultiPlayerGame` abstract class follows:

```
import java.util.List;
import java.util.ArrayList;

public abstract class MultiPlayerGame implements Game {
    private List<Player> players;

    public MultiPlayerGame () {
        init();
        start();
        play();
        end();
    }

    @Override
    public void init() {
        players = new ArrayList<>();
    }

    public void addPlayer(Player player) {
        players.add(player);
    }

    public void removePlayer(Player player) {
        players.add(player);
    }

    public Player getPlayer(int id)  throws PlayerException {
        if (id < 0) {
            throw new PlayerException("The player ID cannot be negative");
        } else if (id >=players.size()) {
            throw new PlayerException("The player ID does not exist");
        }
        return players.get(id);
    }

    public Player getPlayer(String name) throws PlayerException {
        Player playerFound = null;
        int counter = 0;
        for (Player player : players) {
            if (player.getName().equals(name)) {
```

```
                counter++;
                playerFound = player;
            }
        }
        if (counter == 0) {
            throw new PlayerException("Player with name " + name
                + " not found");
        } else if (counter > 1) {
            throw new PlayerException("Found " + counter
                + " players with name " + name);
        }
        return playerFound;
    }

    public List<Player> getPlayers() {
        return players;
    }
}
```

Then we defined its implementation in a minimal way:

```java
import java.util.Iterator;

import java.util.Iterator;

public class MultiPlayerGameImpl extends MultiPlayerGame {
    public void start() {
        // so far, no implementation
    }

    public void play() {
        // so far, no implementation
    }

    public void end() {
        // so far, no implementation
    }

    public void printPlayers() {
        System.out.println("Players list:");
        for (Player player : getPlayers()) {
            System.out.println(player);
        }
    }
}
```

Finally, follows our test class:

```java
public class MultiPlayerGameTest {
    private static int counter;
```

```java
public static void main(String args[]) {
    MultiPlayerGameImpl game = new MultiPlayerGameImpl();
    Player p1 = new Player(++counter,"John");
    Player p2 = new Player(++counter,"James");
    game.addPlayer(p1);
    game.addPlayer(p2);
    game.printPlayers();
    System.out.println();
    //NAME OK
    try {
        System.out.println("Retrieved player: " + game.getPlayer("John"));
    } catch (Exception exc) {
        assert false;
        System.out.println(exc.getMessage());
    }
    //NAME NOT OK
    try {
        System.out.println("Retrieved player: " + game.getPlayer("Jack"));
        assert false;
    } catch (Exception exc) {
        System.out.println(exc.getMessage());
    }
    try {
        Player p3 = new Player(++counter,"John");
        game.addPlayer(p3);
        game.printPlayers();
        System.out.println();
        System.out.println("Retrieved player: " +game.getPlayer("John"));
        assert false;
    } catch (Exception exc) {
        System.out.println(exc.getMessage());
    }
    //ID OK
    try {
        System.out.println("Retrieved player: " +game.getPlayer(2));
    } catch (Exception exc) {
        assert false;
        System.out.println(exc.getMessage());
    }
    //ID NOT OK
    try {
        System.out.println("Retrieved player: " +game.getPlayer(-1));
        assert false;
    } catch (Exception exc) {
        System.out.println(exc.getMessage());
    }
    try {
        System.out.println("Retrieved player: " +game.getPlayer(3));
        assert false;
```

345

```
            } catch (Exception exc) {
                System.out.println(exc.getMessage());
            }
        }
    }
```

which produces the following output:

```
Players list:
Player 1: John
Player 2: James

Retrieved player: Player 1: John
Player exception: Player with name Jack not found
Players list:
Player 1: John
Player 2: James
Player 3: John

Player exception: Found 2 players with name John
Retrieved player: Player 3: John
Player exception: The player ID cannot be negative
Player exception: The player ID does not exist
```

Solution 10.x)

A possible solution is represented by the following code:

```
import java.util.Scanner;

public class MultiPlayerGameConfigurator implements Configurator {
    private static int counter;
    private Scanner scanner;
    private MultiPlayerGameImpl game;

    public MultiPlayerGameConfigurator() {
        game = new MultiPlayerGameImpl();
        scanner = new Scanner(System.in);
        config();
        game.printPlayers();
    }

    @Override
    public void config() {
        var players = game.getPlayers();
        System.out.println("Enter player name " + (++counter) +
            (players.isEmpty() ?"":" Or 'i' to start playing"));
        String text = scanner.next();
        if (!players.isEmpty() && text.equals("i")) {
```

```
                return;
            }
        game.addPlayer(new Player (counter, text));
        config();
    }
}
```

Note the recursive implementation of the `config()` method, which allows us to add as many players as we want.

Solution 10.y)

For completeness, we report the `Player` class which has been made abstract and modified with the addition of the `play()` abstract method (changes in bold) :

```
public abstract class Player {
    private int id;
    private String name;

    public Player(int id, String name) {
        this.setId(id);
        this.setName(name);
    }

    public abstract boolean play();

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Player " + id + ": " + name;
    }
}
```

then follows the implementation of the `GuessNumberPlayer` class, which defines the logic to play in within its `play()` method, very similar to what was the logic in the `GuessNumberGame` version of exercise 10.v:

```java
import java.util.Scanner;

public class GuessNumberPlayer extends Player {
    private Scanner scanner;
    private int numberToGuess;
    private int attemptsNumber;

    public GuessNumberPlayer (int id, String name, Scanner scanner,
            int numberToGuess) {
        super(id, name);
        this.scanner = scanner;
        this.numberToGuess = numberToGuess;
    }

    public void incrementAttemptsNumber() {
        attemptsNumber++;
    }

    public int getAttemptsNumber() {
        return attemptsNumber;
    }

    @Override
    public boolean play() {
        if (scanner.hasNextInt()) {
            incrementAttemptsNumber();
            int number = scanner.nextInt();
            if (number < numberToGuess) {
                System.out.println("Too low, try again");
            } else if (number > numberToGuess) {
                System.out.println("Too high, try again");
            } else {
                return true;
            }
        } else {
            System.out.println("Invalid input '" + scanner.next ()
                + "'. You can only enter integer numbers, please try again");
            play();
        }
        return false;
    }
}
```

Finally, below is our implementation of the new GuessNumberGame class (changes in bold):

```java
import java.util.Scanner;
import java.util.Random;
import java.util.List;
import java.util.ArrayList;

public class GuessNumberGame extends MultiPlayerGame implements Configurator {
```

```java
    private static final int MAX_NUMBER = 100;
    private static int counter;
    private Player currentPlayer;
    private Scanner scanner;
//    private int attemptsNumber;
    private int numberToGuess;

/*
    public GuessNumberGame () {
        init();
        start();
        play();
        end();
    }
*/

    @Override
    public void init() {
        super.init();
        scanner = new Scanner(System.in);
        numberToGuess = GuessNumber.generateRandomNumber(MAX_NUMBER);
    }

    @Override
    public void start() {
        config();
        System.out.println("I'm thinking of a number between 1 and "
            + MAX_NUMBER + ", guess what!");
    }

    @Override
    public void config() {
        System.out.println("Enter player name " + (++counter) +
            (getPlayers().isEmpty() ?"":" Or 'i' to start playing"));
        String text = scanner.next();
        if (!getPlayers().isEmpty() && text.equals("i")) {
            currentPlayer = getPlayers().get(0);
            System.out.println(currentPlayer + " is your turn");
            return;
        }
        addPlayer(new GuessNumberPlayer (counter, text, scanner,
            numberToGuess));
        config();
    }

    @Override
    public void play() {
        while (scanner.hasNext()) {
            boolean result = currentPlayer.play();
            if (result) {
                return;
```

```
            }
            currentPlayer = getCurrentPlayer();
            System.out.println(currentPlayer + " is your turn");
        }
    }

    private Player getCurrentPlayer() {
        int nextPlayerIndex = currentPlayer.getId();
        var players = getPlayers();
        if (nextPlayerIndex == players.size()) {
            nextPlayerIndex = 0;
        }
        return players.get(nextPlayerIndex);
    }

    @Override
    public void end() {
        System.out.println("Yes " + currentPlayer.getName()
            + "! You guessed it after " + ((GuessNumberPlayer)currentPlayer)
            .getAttemptsNumber() + " attempts");
    }

    public static void main(String args[]) {
        new GuessNumberGame();
    }
}
```

Solution 10.z)

The solution should be:

1. The PlayerException class belongs to games.generic.exceptions.

2. The Player and MultiPlayerGame classes belong to games.generic.data.

3. The GuessNumberPlayer class belongs to games.guessnumber.data.

4. The GuessNumberGame class belongs to games.guessnumber.business.

5. The GuessNumber class belongs to games.guessnumber.util.

6. The Game class belongs to games.generic.business.

7. The Configurator class belongs to games.generic.business.

Note that all classes belonging to the games.generic package are classes that could be reused in other games! So, in practice we created, what in architectural language is called a *framework*, let's say a *mini-framework*, which we can reuse to create new configurable and multi-user games!

Chapter 11
Exercises

Enumerations and Nested Types

Chapter 11 is the first of the chapters of the third part of the book named "Advanced Features". In particular, the nested types are one of the most complex arguments of Java, due to their syntax, and above all to the abstruse properties that characterize them. Mastering the nested types will allow us to interface with the language more effectively. As for the enumerations, all in all it is a topic not so complex. There are some particular rules that need to be understood, but the important thing is to understand when it is convenient to use them. Also for this chapter, you will find many exercises with multiple answers (some very difficult to solve) that support the preparation of the Oracle certification exams, but also implementations to be coded.

Exercise 11.a) Nested Types, True or False:

1. A nested class is a class that is declared within another class.

2. An anonymous class is also nested, but has no name. Moreover, to be declared, it must necessarily be instantiated.

3. The nested classes are not necessary for the Object Orientation.

4. A nested class must necessarily be instantiated.

5. To instantiate a nested public class, sometimes the external class must be instantiated first.

6. A private nested class, must also declare the "set" and "get" methods to be used by a third class.

7. A nested class cannot have the same name as the class that contains it.

8. An anonymous class can have the same name as the class that contains it.

9. A nested class can access static members of the containing class only if it is declared static.

10. A nested class cannot be declared abstract.

Exercise 11.b) Enumerations, True or False:

1. The enumerations objects cannot be instantiated except within the definition of the enumeration itself. In fact, they can only have private constructors.

2. An enumeration can declare methods, and its elements can be extended by classes which can override the enumeration methods. However, it is not possible for one enum extends another enum.

3. The `values()` method belongs to every enumeration but not to the `java.lang.Enum` class.

4. The following code is compiled without errors:

```
public enum MyEnum {
  public void method1() {

  }
  public void method2() {

  }
  ENUM1, ENUM2;
}
```

5. The following code is compiled without errors:

```
public enum MyEnum {
  ENUM1 {
    public void method() {

    }
  }, ENUM2;
  public void method2() {

  }
}
```

6. The following code is compiled without errors:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  private MyEnum(int i) {

  }
}
```

7. The following code is compiled without errors:

```
public class Volume {
  public enum Level {
    HIGH, MEDIUM, LOW
  };
  // class implementation. . .
  public static void main(String args[]) {
    switch (getLevel()) {
      case HIGH:
          System.out.println(Level.HIGH);
      break;
      case MEDIUM:
          System.out.println(Level.MEDIUM);
      break;
      case LOW:
          System.out.println(Level.LOW);
      break;
    }
  }
  public static Livello getLevel() {
    return Level.HIGH;
  }
}
```

8. If we declare the following enumeration:

```
public enum MyEnum {
  ENUM1 {
    public void method1() {

    }
  },
  ENUM2 {
    public void method2() {

    }
  }
}
```

the following code could be correctly compiled:

```
MyEnum.ENUM1.method1();
```

9. It is not possible to declare enumerations with a single element.

10. Enumerations can be nested into enumerations in the following way:

```
public enum MyEnum {
  ENUM1 (), ENUM2;
  public enum MyEnum2 {a,b,c}
}
```

and the following code is compiled without errors:

```
System.out.println(MyEnum.MyEnum2.a);
```

Exercise 11.c)

Modify the sources created with the exercises of Chapter 8, after replacing the `FileType` interface, created in exercise 8.c, with an enumeration. Everything will have to work as before.

Exercise 11.d)

Starting from the solution of the previous exercise, insert the `FileType` enumeration in the `File` class. What changes must be made to the application to continue to make it work as before?

Exercise 11.e)

Consider the sources created with the exercises of Chapter 6, and modified with the exercises in Chapter 9. Currently, the `Coin` class may be instantiated incorrectly, and exceptions may need to be handled. For example, as a parameter to the constructor of the `Coin` class a negative value could be passed. Following is the code we have developed so far (comments omitted):

```
public class Coin {

    public final static String CURRENCY ="EURO";

    private final int value;

    public Coin(int value) {
        this.value = value;
        System.out.println("Created a " + getDescription());
    }

    private static String formatDescriptiveString(int value) {
        String formattedString = " cents of ";
        if (value == 1) {
            formattedString = " cent of ";
```

```
        } else if (value > 99) {
            formattedString = " ";
            value /= 100;
        }
        return value + formattedString;
    }

    public int getValue() {
        return value;
    }

    public String getDescription() {
        String description = "coin of " + formatDescriptiveString(value)
                                + CURRENCY;
        return description;
    }
}
```

Create an enumeration that allows to make the constructor (and therefore the entire program) more robust, and that allows us to avoid exception handling. Rewrite also the `Coin` class.

Exercise 11.f)

Based on the previous exercise, change the `Purse` class accordingly.

Exercise 11.g)

Based on the last two exercises, change the `CoinsTest` class accordingly.

Exercise 11.h)

Suppose we want to write a program and want to re-use the following `Person` class, inherited from a program already written and not editable:

```
public class Person {

    private String name;
    private String surname;
    private String birthDate;
    private String occupation;
    private String address;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
```

355

```java
    public Person(String name, String surname, String birthDate,
            String occupation, String address) {
        this.name = name;
        this.surname = surname;
        this.birthDate = birthDate;
        this.occupation = occupation;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public String getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(String birthDate) {
        this.birthDate = birthDate;
    }

    public String getOccupation() {
        return occupation;
    }

    public void setOccupation(String occupation) {
        this.occupation = occupation;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
```

356

```
    @Override
    public String toString() {
        return "Person{" + "name=" + name + ", surname=" + surname + '}';
    }
}
```

Unfortunately, in our context, we would need to redefine the `toString()` method so that it not only prints information on the person's first and last name, but also the date of birth, address and occupation. As already mentioned, however, the class is already in use and cannot be changed. In particular, our requirement is that the `toString()` method returns the following string:

```
Name:               Arjen Anthony
Surname:            Lucassen
Occupation:    Composer
Birth Date     03/04/1960
Address:            Holland
```

Then create a `PersonTest` class that redefines the `toString()` method of the `Person` class and prints the above output.

> For formatting it is possible to exploit the escape character `\t` introduced in paragraph 3.3.5.3.

Exercise 11.i)

Given the following class:

```java
public class External {
    private int integer = 5;

    public static void main(String[] args) {
        External.Inner inner = new External().new Inner();
        inner.innerMethod();
    }
    private class Inner {
        private int integer = 10;
        protected void innerMethod() {
            System.out.println(super.integer);
        }
    }
}
```

Which of the following statements are true?

1. Once the `External` class has been executed, a `NullPointerException` is launched.

2. Once the `External` class has been executed, `50` is printed.

3. Once the `External` class has been executed, `10` is printed.

4. Once the `Internal` class has been executed, `5` is printed.

5. Once the `Internal` class has been executed, `0` is printed.

6. This code cannot be compiled.

Exercise 11.l)

Taking into account the answer of the previous exercise, modify the code (with a minimal change) so as to print the value 5. Then modify it again to make it print also the value 10.

Exercise 11.m)

Given the following class:

```java
public class External {
    public int a = 1;
    private int b = 2;
    public void externalMethod(final int c) {
        int d = 4;
        class Inner {
            private void innerMethod(int e) {

            }
        }
    }
}
```

Which of the following statements are correct?

1. Within the method `innerMethod` it is possible to reference the variable a.

2. Within the method `innerMethod` it is possible to reference the variable b.

3. Within the method `innerMethod` it is possible to reference the variable c.

4. Within the method `innerMethod` it is possible to reference the variable d.

5. Within the method `innerMethod` it is possible to reference the variable e.

6. The class cannot be compiled.

Exercise 11.n)

Given the following class:

```
public class Exercise11N {

    public static void main(String args[]) {
        new MyInterface(){
            public void method(){
                System.out.println("Anonymous class");
            }
        }.method();
    }

    private interface MyInterface{
        void method();
    }
}
```

Which of the following statements is true?

1. Executing the `Exercise11N` class, a `NullPointerException` is thrown.

2. Executing the `Exercise11N` class, a `CannotInstantiateInterfaceException` is thrown.

3. Once the `Exercise11N` class has been executed, "Anonymous Class" is printed.

4. Once the `Exercise11N` class has been executed, "Null" is printed.

5. This code cannot be compiled.

Exercise 11.o)

Given the following class:

```
public class External {
    private class Inner {
        private static int effectivelyFinalVariable = 10;
        Inner() {
            effectivelyFinalVariable = 11;
        }
        protected void method() {
            System.out.println(effectivelyFinalVariable);
        }
    }
}
```

Which of the following statements are true?

1. This code cannot be compiled.

2. The `effectivelyFinalVariable` variable is not "effectively final".

3. The `effectivelyFinalVariable` variable does not matter whether or not it is "effectively final" because it is an instance variable.

4. The `effectivelyFinalVariable` variable does not matter whether it is "effectively final" or not because it belongs to the internal class and not to the external class.

Exercise 11.p)

Given the following class:

```java
public class External {
    private final static String string = "Nested class";

    protected External() {
        private static class Nested {
            protected void method() {
                System.out.println(string);
            }
        }
    }
}
```

Which of the following statements are true?

1. This code cannot be compiled.

2. The static `string` constant is not "effectively final" because it is also static, and therefore cannot be used within the internal class `Nested`.

3. The static string constant is not accessible to the `method()` method because it is declared static.

4. The `Nested` class, being declared inside a protected constructor, can only be used inside the constructor.

Exercise 11.q)

Given the following code:

```java
public enum Exercise11Q {
    A, B, C;
```

```
    private enum InnerEnum {
        C, D, E;

        protected enum InnerInnerEnum {
            F,G,H
        }
    }


    public static void main(String args[]) {
        System.out.println(/*INSERT YOUR CODE HERE*/);
    }
}
```

Which of the following expressions can be inserted instead of the comment /* INSERT YOUR CODE HERE */ to print the value H (it is possible to choose zero or more expressions):

1. `Exercise11Q.A.InnerEnum.C.InnerInnerEnum.H`

2. `Exercise11Q.InnerEnum.InnerInnerEnum.H`

3. `InnerInnerEnum.H`

4. `InnerEnum.A.InnerInnerEnum.H`

5. `InnerEnum.InnerInnerEnum.F.G.H`

6. `InnerEnum.InnerInnerEnum.H`

7. `Exercise11Q.A.C.H`

8. None: the code does not compile.

Given the following code:

```
public enum Exercise11R implements Interface {
    ONE {
        @Override
        public int method() {
            return 29 + 7 + 74;
        }
    } ,
    TWO,
    THREE{
        @Override
```

```
            public int method() {
                return 12 + 11 + 6;
            }
        };
        @Override
        public int method() {
            return 14 + 4 + 4;
        }

        public static void main(String args[]) {
            Interface i = Exercise11R.THREE;
            System.out.println(i.method());
        }
    }

    interface Interface {
        int method();
    }
```

If the `Exercise11R` file is executed, what will be printed (choose only one answer)?

1. 22

2. 29

3. 110

4. 161

5. THREE

6. THREE.method()

7. i.method()

8. None: the code does not compile.

Given the following code:

```
public enum Exercise11S {
    A, B, C;
    public class Inner{
        public enum InnerEnum {
            D, E, F;
        }
    }
    public static void main(String args[]) {
        for (Exercise11S.Inner.InnerEnum item : Exercise11S.Inner.InnerEnum.values()) {
```

```
                    System.out.println(item);
            }
        }
}
```

Once the `Exercise11S` enumeration has been performed, the output will be:

1. A, B, C

2. D, E, F

3. A B C

4. D E F

5. ABC

6. DEF

7. None of the above: a `NullPointerException` is thrown.

8. None of the above: the code does not compile due to an error in the `main()` method.

9. None of the above: the code does not compile due to an error in the `Inner` class.

Exercise 11.t)

> **(SPOILER ALERT: next lines will reveal the solution of the previous exercise!)**

After reading the solution of the previous exercise (in particular the compilation output), modify the code to make it compile, solving the various errors that occur after the various compilations. Running the definitive compilable version the program will have to print the following output:

```
D
E
F
```

> **This exercise tests your ability to interpret compiler error messages, and solve them. Practically what every programmer does all the time!**

363

Exercise 11.u)

What should we use to have different (even partial) algorithms run on a method (choose all the correct answers) without having first created the algorithm code?

1. An object instantiated from an existing class.

2. An interface.

3. An enumeration.

4. An inner class.

5. A nested class.

6. An anonymous class.

7. A switch construct.

8. Nothing, is simply not possible.

Exercise 11.v)

Abstract a Neapolitan deck of cards and create an executable class (with the `main()` method that runs through all 40 cards.

Exercise 11.z)

Starting from the solution of the previous exercise:

1. Identify formatting problems.

2. Fix the formatting problems identified.

> **This exercise tests your ability to solve problems with inventiveness, conscience and initiative. Practically what every programmer should always do!**

Chapter 11
Exercise Solutions

Enumerations and Nested Types

1. **True**.

2. **True**.

3. **True**.

4. **False**, anonymous classes must necessarily be instantiated.

5. **True**, see section 11.1.2.

6. **False**.

7. **True**.

8. **False**, an anonymous class has no name.

9. **True**.

10. **True**.

11. **False**, an anonymous class cannot be declared abstract.

Solution 11.b) Enumerazioni, True or False:

1. **True**.

2. **True**.

3. **False**.

4. **False**.

5. **True**.

6. **False**, it is not possible to use the default constructor if one is explicitly declared.

7. **True**.

8. **True**.

9. **True**.

10. **True**.

Solution 11.c)

The `FileType` enumeration code is very simple:

```
public enum FileType {
    JAVA,
    C_SHARP,
    C_PLUS_PLUS,
    C;
}
```

Then you need to modify the `File` class, so as to use the `FileType` type instead of the old integer type:

```
public abstract class File {

    private String name;

    private FileType type;

    public File(String name, FileType type) {
        this.name = name;
        this.type = type;
    }

    public String getName() {
        return name;
    }
```

```java
    public void setName(String name) {
        this.name = name;
    }

    public FileType getType() {
        return type;
    }

    public void setType(FileType type) {
        this.type = type;
    }
}
```

Then you need to modify the SourceFile class, so as to use the FileType type instead of the old integer type:

```java
public class SourceFile extends File {
    private String content;

    public SourceFile(String name, FileType type) {
        super(name, type);
    }

    public SourceFile(String name, FileType type, String content) {
        this(name, type);
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public void addText(String text) {
        if (content == null) {
            content ="";
        }
        if (text != null) {
            content += text;
        }
    }

    public void addText(String text, int position) {
        final int length = content.length();
        if (content != null && text != null && position > 0 &&
        position < length) {
```

```
                content = content.substring(0, position) + text +
                content.substring(position);
        }
    }
}
```

Everything else can remain as it is.

Solution 11.d)

The `File` code with the `TypeFile` nested enumeration should be the following:

```
public abstract class File {
    public enum FileType {
        JAVA,
        C_SHARP,
        C_PLUS_PLUS,
        C;
    }

    private String name;

    private FileType type;

    public File(String name, FileType type) {
        this.name = name;
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public FileType getType() {
        return type;
    }

    public void setType(FileType type) {
        this.type = type;
    }
}
```

Then we need to modify the two test classes to correctly reference the `FileType` nested element:

368

```
public class SourceFileTest {
    public static void main(String args[]) {
        SourceFile sourceFile = new SourceFile("Test.java",
            File.FileType.JAVA, "public class MyClass {\n\r");
        System.out.println(sourceFile.getContent());
        // Correct test addText (String)
        sourceFile.addText("}");
        System.out.println(sourceFile.getContent());
        // Correct test addText (String,int)
        sourceFile.addText("//Test text addition\n\r", 23);
        System.out.println(sourceFile.getContent());
        // Incorrect test addText (String,int)
        sourceFile.addText("//Test text addition\n\r", -1);
        System.out.println(sourceFile.getContent());
        // Incorrect test addText (String,int)
        sourceFile.addText("//Test text addition\n\r", 100);
        System.out.println(sourceFile.getContent());
    }
}
```

and:

```
public class IDETest {
    public static void main(String args[]) {
        IDE ide = new JavaIDE();
        SourceFile sourceFile = new SourceFile("Test.java",
            File.FileType.JAVA, "public class MyClass {\n\r");
        ide.update(sourceFile, "}");
    }
}
```

Instead, it was not necessary to modify the SourceFile class, because being a File subclass it inherited the enumeration.

Solution 11.e)

The implementation of our enumeration is quite complex:

```
public enum Value {

    ONE_CENT(1) {
                @Override
                public String getDescriptiveString() {
                    return getValue() + " cents of ";
                }
            },
    TWO_CENTS(2),
    FIVE_CENTS(5),
    TEN_CENTS(10),
```

```
        TWENTY_CENTS(20),
        FIFTY_CENTS(50),
        ONE_EURO(1) {
                    @Override
                    public String getDescriptiveString() {
                        return getValue() + " ";
                    }
                },
        TWO_EURO(2) {
                    @Override
                    public String getDescriptiveString() {
                        return getValue() + " ";
                    }
                };

        private int value;

        private Value(int value) {
            this.value = value;
        }

        public String getDescriptiveString() {
            return getValue() + " cents of ";
        }

        public int getValue() {
            return value;
        }
    }
```

We have defined all the possible values for a coin (supposed to be a Euro). Each element of the enumeration uses a constructor that sets the numerical value of the `value` variable. Furthermore, the `getDescriptiveString()` method is defined, which will simplify the original `Coin` class. This method is overridden by anonymous classes for three elements of the enumeration. The `Coin` class must be modified (simplified) as follows:

```
public class Coin {

    public final static String CURRENCY = "EURO";

    private final Value value;

    public Coin(Value value) {
        this.value = value;
        System.out.println("Created a " + getDescription());
    }

    public Value getValue() {
```

```
            return value;
    }

    public String getDescription() {
        String description = "coin value " + value.getDescriptiveString()
                + CURRENCY;
        return description;
    }
}
```

Solution 11.f)

The Purse class code must be modified as follows (comments omitted):

```
public class Purse {

    private static final int DIMENSION = 10;
    private final Coin[] coins = new Coin[DIMENSION];

    public Purse(Value... values) {
        assert coins.length == DIMENSION;
        try {
            int numberOfCoins = values.length;
            for (int i = 0; i < numberOfCoins; i++) {
                if (i >= 10) {
                    throw new FullPurseException (
                        "Only the first 10 coins have been inserted!");
                }
                coins[i] = new Coin(values[i]);
            }
//  } catch (FullPurseException | NullPointerException exc) {
        } catch (FullPurseException exc) {
            System.out.println(exc.getMessage());
        } catch (NullPointerException exc) {
            System.out.println("The purse has been created empty");
        }
        assert coins.length == DIMENSION;
    }

    public void add(Coin coin) throws FullPurseException {
        try {
            System.out.println("Let's try adding one "+
            coin.getDescription());
        } catch (NullPointerException exc) {
            throw new NullCoinException();
        }
        int freeIndex = firstFreeIndex();
```

```
        if (freeIndex == -1) {
            throw new FullPurseException("Purse full! The coin "
            + coin.getDescription() + " has not been added!");
        } else {
            coins[freeIndex] = coin;
            System.out.println(coin.getDescription() +" has been added");
        }
    }

    public Coin withdraw(Coin coin) throws CoinNotFoundException {
        try {
            System.out.println("Let's try to get a "+
                    coin.getDescription());
        } catch (NullPointerException exc) {
            throw new NullCoinException();
        }
        Coin foundCoin = null;
        int foundCoinIndex = foundCoinIndex(coin);
        if (foundCoinIndex == -1) {
            throw new CoinNotFoundException("Coin not found!");
        } else {
            foundCoin = coin;
            coins[foundCoinIndex] = null;
            System.out.println("One " + coin.getDescription() + " withdraw");
        }
        return foundCoin;
    }

    public void state() {
        System.out.println("The purse contains:");
        for (Coin coin : coins) {
            if (coin == null) {
                break;
            }
            System.out.println("One "+ coin.getDescription());
        }
    }

    private int firstFreeIndex() {
        int index = -1;
        for (int i = 0; i < 10; i++) {
            if (coins[i] == null) {
                index = i;
                break;
            }
        }
        return index;
    }
```

```
      private int foundCoinIndex(Coin coin) {
          int foundCoinIndex = -1;
          for (int i = 0; i < 10; i++) {
              if (coins[i] == null) {
                  continue;
              }
              Value coinInPurseValue = coins[i].getValue();
              Value value = coin.getValue();
              if (value == coinInPurseValue) {
                  foundCoinIndex = i;
                  break;
              }
          }
          return foundCoinIndex;
      }
  }
```

Solution 11.g)

The CoinsTest class code is as follows:

```
public class CoinsTest {
    public static void main(String args[]) {
        Coin twentyCentsCoin = new Coin(Value.TWENTY_CENTS);
        Coin oneCentCoin = new Coin(Value.ONE_CENT);
        Coin oneEuroCoin = new Coin(Value.ONE_EURO);
        // Creation of a Purse with 11 coins
        Purse purseToFail = new Purse(Value.TWO_CENTS, Value.FIVE_CENTS,
                                      Value.ONE_EURO, Value.TEN_CENTS,
                                      Value.FIFTY_CENTS, Value.TEN_CENTS,
                                      Value.ONE_EURO, Value.TWO_EURO,
                                      Value.TEN_CENTS, Value.FIVE_CENTS,
                                      Value.TWO_CENTS);
        // Creation of a Purse with 8 coins
        Purse purse = new Purse(Value.TWO_CENTS, Value.FIVE_CENTS,
                                Value.ONE_EURO, Value.TEN_CENTS,
                                Value.FIFTY_CENTS, Value.TEN_CENTS,
                                Value.ONE_EURO, Value.TWO_EURO);
        purse.state();

        try {
            // we add a 20 cents coin
            purse.add(twentyCentsCoin);
        } catch (FullPurseException | NullCoinException exc) {
            System.out.println(exc.getMessage());
        }

        try {
            // we add a 1 cents coin
```

373

```
                    purse.add(oneCentCoin);
            } catch (FullPurseException | NullCoinException exc) {
                System.out.println(exc.getMessage());
            }

            try {
                // We add the eleventh coin (we should get an error and the
            // coin will not be added)
                purse.add(oneEuroCoin);
            } catch (FullPurseException | NullCoinException exc) {
                System.out.println(exc.getMessage());
            }

            // We evaluate the status of the purse
            purse.state();

            try {
                // we withdraw 20 cents
                purse.withdraw(twentyCentsCoin);
            } catch (CoinNotFoundException exc) {
                System.out.println(exc.getMessage());
            }

            try {
                //Let's add the tenth coin again
                purse.add(oneEuroCoin);
            } catch (FullPurseException | NullCoinException exc) {
                System.out.println(exc.getMessage());
            }

            // We evaluate the status of the purse
            purse.state();

    //The next example has no sense now!
    //          try {
    //          // We withdraw a non-existent currency (we should get an error)
    //              purse.withdraw(new Coin(7));
    //          } catch (CoinNotFoundException exc) {
    //              System.out.println(exc.getMessage());
    //          }

            try {
                //We try to add null
                purse.add(null);
            } catch (FullPurseException | NullCoinException exc) {
                System.out.println(exc.getMessage());
            }

            try {
```

374

```
                //We try to withdraw null
                purse.withdraw(null);
            } catch (CoinNotFoundException | NullCoinException exc) {
                System.out.println(exc.getMessage());
            }
            //we test the passage of the null value to the purse constructor
            Purse purseWithNullPointerException = new Purse(null);
            purse.state();
        }
    }
```

Solution 11.h)

The solution is very simple using an anonymous class on the fly as in the following code, but we must format the code using tabs ("\t"):

```java
public class PersonaTest {
    public static void main(String args[]) {
        System.out.println(
          new Person("Arjen Anthony", "Lucassen","03/04/1960", "Composer",
              "Holland") {
            @Override
            public String toString() {
                String string ="Name: \t\t\t"+ getName();
                string +="\nSurname: \t\t"+ getSurname();
                string +="\nOccupation: \t\t"+ getOccupation();
                string +="\nBirth Date \t\t"+ getBirthDate();
                string +="\nAddress: \t\t"+ getAddress();
                return string;
            }
        } );
    }
}
```

Solution 11.i)

The correct answer is the last one. In fact the super.integer expression would imply that the superclass of the Inner class has a variable called integer, which actually does not exist. Here is the compile-time error message:

```
External.java:11: error: cannot find symbol
          System.out.println(super.integer);
                                  ^
  symbol: variable integer
1 error
```

Solution 11.l)

The solution could be the following:

```
public class External {
    private int integer = 5;

    public static void main(String[] args) {
        External.Inner inner = new External().new Inner();
        inner.innerMethod();
    }
    private class Inner extends External {
        private int integer = 10;
        protected void innerMethod() {
            System.out.println(super.integer);
            System.out.println(this.integer);
        }
    }
}
```

We have highlighted the changes made in bold.

Solution 11.m)

The correct statements are 1, 2, 3, 4 and 5.

Solution 11.n)

The correct statement is 3. In fact, the `main()` method code instantiates an anonymous class on the fly, redefines the `method()` method and, without assigning the new instance to a reference, calls (always on the fly) the method just redefined.

It would also have been advisable to annotate the redefined method with the `Override` annotation.

Solution 11.o)

All statements are true. In particular, the number 1 is true, because rule number 7 applies (see section 11.1.2.5): "A nested class can declare static members only if declared static.".

376

Solution 11.p)

Statement 1 is correct. In fact, it is not possible to compile this code, because it is not possible to declare a nested class inside a constructor (or a method).

The number 2 is false because, while the first statement could be considered correct (if it is declared static, a variable cannot be local, and only the local variables can actually be final), the second is not true because the `Nested` nested class is a in turn declared static, and therefore could use the static variables of the `External` class.

For the same reason, the number 3 is also false.

The number 4 is obviously false because, as we said for statement 1, it is not possible to declare a nested class within a constructor (or a method).

Solution 11.q)

The correct answers are 2, 4 and even 5.

Solution 11.r)

The right answer is 2, which means that it is printed 29. In fact, the `method()` method, inherited from the `InnerInterface` interface, is rewritten by the enumeration `Exercise11R` with an implementation that returns the number 22. But for the specific element of the enumeration `THREE` , the method is rewritten in such a way as to precisely return the value 29. In the `main()` method the `Exercise11R.THREE` element is assigned to the reference `i` of the `Interface` interface (it is legal for data polymorphism). Then the `method()` method is invoked on `i`, which actually points to the `THREE` element, which, as we have said, has redefined an implementation of the `method()` method, which returns 29.

Solution 11.s)

The correct answer is the last one (number 9). In fact, it is not possible to declare an enumeration within an inner class (see section 11.3.2).

The output of the compilation is as follows:

```
Exercise11S.java:4: error: enum declarations allowed only in static contexts
        public enum InnerEnum {
                ^
1 error
```

Solution 11.t)

Interpreting the output of the previous solution:

```
Esercizio11S.java:4: error: enum declarations allowed only in static
  contexts
        public enum InnerEnum {
                ^
1 error
```

It is easy to understand that the problem is that the inner classes (which we remember to be non-static nested classes) cannot declare enumerations. The compiler, however, tells us that enumeration declarations are allowed only in static contexts, so it is natural to declare the nested class as static in the following way:

```
public enum Exercise11T {
    A, B, C;
    public static class Inner {
        public enum InnerEnum {
            D, E, F;
        }
    }
    public static void main(String args[]) {
        for (Exercise11T.Inner.InnerEnum item : Exercise11T.Inner.InnerEnum.values()) {
            System.out.println(item);
        }
    }
}
```

Compiling this code, we will get the following output:

```
Exercise11T.java:9: error: package Exercise11T.Inner does not exist
    for (Exercise11T.Inner.InnerEnum item :
 Exercise11T.Inner.InnerEnum.values()) {
            ^
Exercise11T.java:9: error: package Exercise11T.Inner does not exist
    for (Exercise11T.Inner.InnerEnum item : Exercise11T.Inner.InnerEnum.values()) {
                        ^
2 errors
```

Which makes us understand that the type `Exercise11T.Inner.InnerEnum` is not recognized as a valid type. The solution consists in using the correct type for our visibility context: `Inner.InnerEnum` (that is, we have not specified the enumeration that contains both the `main()` method and the `Inner` class). So the code should look like this:

```
public enum Exercise10T {
    A, B, C;
    public static class Inner {
```

```
        public enum InnerEnum {
            D, E, F;
        }
    }
    public static void main(String args[]) {
        for (Inner.InnerEnum item : Inner.InnerEnum.values()) {
            System.out.println(item);
        }
    }
}
```

which will be compiled without errors, and whose output will be the following:

```
D
E
F
```

as requested.

Solution 11.u)

The answer is number 6, as you can check in section 11.2.5 when in the video store example we had created the code to be executed on the fly by a method with the following syntax:

```
public class VideoStoreTest {
    public static void main(String args[]) {
//...
        System.out.println("Nice movies:");
        Movie[] niceMovies =  videoStore.getFilteredMovies(new MovieFilter() {
            public boolean filter(Movie movie) {
                return movie.getReviewsAverage() > 3;
            }
        });
//...
        System.out.println("\nSciFi movies:");
        Movie[] sciFiMovies =  videoStore.getFilteredMovies(new MovieFilter() {
            public boolean filter(Movie movie) {
                return "SciFi".equals(movie.getGenre());
            }
        } );
//...
    }
}
```

where the VideoStore was:

```
public class VideoStore {
    private Movie[] movies;
```

```
    public VideoStore() {
        movies = new Movie[10];
        loadMovies();
    }
    public void setFilms(Movie[] movies) {
        this.movies = movies;
    }

    public Movie[] getFilms() {
        return movies;
    }

/* THE FOLLOWING METHODS HAVE BEEN REPLACED BY THE DEFINED METHOD IMMEDIATELY AFTER
    public Movie[] getSciFiMovies() {
        Movie [] sciFiMovies = new Movie[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if ("SciFi".equals(movies[i].getGenre())) {
                sciFiMovies[j] = movies[i];
                j++;
            }
        }
        return sciFiMovies;
    }
    public Movie[] getNiceMovies() {
        Movie [] niceMovies = new Movie[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if (movies[i].getReviewsAverage() > 3) {
                niceMovies[j] = movies[i];
                j++;
            }
        }
        return niceMovies;
    }
*/
    /* THIS METHOD REPLACES THE TWO PREVIOUS (COMMENTED OUT) METHODS */
    public Movie[] getFilteredMovies(MovieFilter movieFilter) {
        Movie [] filteredMovies = new Movie[10];
        for (int i = 0, j= 0; i< 10;i++) {
            if (movieFilter.filter(movies[i])) {
                filteredMovies[j] = movies[i];
                j++;
            }
        }
        return filteredMovies;
    }

    private void loadMovies() {
        //loading movies...
    }
}
```

380

and where the MovieFilter interface, which is redefined with the anonymous classes in the VideoStoreTest class, was simply the following:

```
public interface MovieFilter {
    boolean filter(Movie movie);
}
```

We decided to use enumerations both to abstract the concept of seed and the number of cards. A number could be represented with a primitive type as an int, or even a byte. But all in all, for our purpose the enumeration seemed the best choice to be able to also represent the various cards verbatim. Below is the Number enumeration code:

```
public enum Number {

    ONE("Ace"),
    TWO("Two"),
    THREE("Three"),
    FOUR("Four"),
    FIVE("Five"),
    SIX("Six"),
    SEVEN("Seven"),
    EIGHT("Eight"),
    NINE("Nine"),
    TEN("Ten");

    String representation;

    Number(String representation){
        this.representation = representation;
    }

    @Override
    public String toString(){
        return representation;
    }
}
```

Note that we have used the representation variable to manage the text representation of the enumeration elements. This is set at the time of definition using the only constructor supplied. With this variable we could have the string Ace printed instead of One, in a rather simple way (that is without using particular algorithms that use conditions like switch or if). If we had used a primitive type like int instead, we would have had to manage the situation with an algorithm, which implies greater probability of error. Note that we have also redefined the toString()

381

method inherited from the `Object` class to facilitate the printing of enumeration elements.
To be sure we have written the correct code, we have also created a test class, similar to a unit test (see section 10.4.1) simplified, to test only the printing of the elements of the enumeration:

```
/**
 * Class that tests the number enumeration.
 */
public class NumberTest {
    public static void main (String args []) {
        for (Object object: Number.values ()) {
            System.out.println (object);
        }
    }
}
```

Il cui output ha verificato la correttezza del codice (secondo le nostre intenzioni):

```
Ace
Two
Three
Four
Five
Six
Seven
Eight
Nine
Ten
```

Let's move on to the enumeration that represents the Seed of a card:

```
public enum Seed {
    CUPS,
    STICKS,
    COINS,
    SWORDS;

    String representation;

    public String toString() {
        return doCapitalization(this.name());
    }

    private String doCapitalization(String string) {
        //make the string lowercase
        String lowerCaseString = string.toLowerCase();
        //retrieve the first character of the string
        String initialCharacter = lowerCaseString.substring(0,1);
        //make uppercase the first character
```

```
        String initialCharacterUpperCase = initialCharacter.toUpperCase();
        //retrieve the concatenation between the capital letter
        // and the rest of the lowercase string
        return  initialCharacterUpperCase + lowerCaseString.substring(1);
    }
}
```

In this case, for educational purposes, we have modified the way in which the elements of the enumeration are represented. We have created a utility method called doCapitalization(), which makes the string passed as input capitalized. The instructions of this method are commented and easily to understand. The toString() method does nothing but return the uppercase string of the enumeration element name. Also in this case we have created a test class:

```
/**
 * Class that tests the Seme enumeration.
 */
public class SeedTest {
    public static void main (String args []) {
        for (Object object: Seed.values ()) {
            System.out.println (object);
        }
    }
}
```

which once executed produces the desired output:

```
Cups
Sticks
Coins
Swords
```

The Card class is very simple:

```
public class Card {

    private Seed seed;
    private Number number;

    public Card (Number number, Seed seed) {
        this.number = number;
        this.seed = seed;
    }

    public void setNumber(Number number) {
        this.number = number;
    }

    public Number getNumber() {
        return number;
    }
```

```
    public void setSeed(Seed seed) {
        this.seed = seed;
    }

    public Seed getSeed() {
        return seed;
    }

    public String toString() {
        return number + " of " + seed;
    }
}
```

The most complicated class is undoubtedly the CardDeck class, which we wanted to abstract with a two-dimensional array of 4 lines (one for each seed) and 10 columns (one for each card number):

```
public class CardsDeck {
    private Card[][] cards;

    public CardsDeck() {
        cards = new Card[4][10];
        prepareCards();
    }

    public void prepareCards() {

        Seed[] seeds = Seed.values();
        Number[] numbers = Number.values();
        int seedsLength = seeds.length;
        int numbersLength = numbers.length;

        for (int i = 0; i < seedsLength; i++) {
            for (int j = 0; j < numbersLength; j++) {
                cards[i][j] = new Card(numbers[j], seeds[i]);
            }
        }
    }

    public String toString() {

        int cardsLength = cards.length;
        String string ="";

        for (int i = 0; i < cardsLength; i++) {
            int cardsLengthRow = cards[i].length;

            for (int j = 0; j < cardsLengthRow; j++) {
                string += cards[i][j] + (j != (cardsLengthRow-1) ? ", " : "");
            }
```

```
                    if (i != (cardsLength-1)) {
                        string+="\n";
                    }
            }
            return string;
        }

        public void setCards(Card[][] cards) {
            this.cards = cards;
        }

        public Card[][] getCards() {
            return cards;
        }
}
```

Note that the constructor instantiates the single instance variable (i.e. the two-dimensional array of cards) and then invokes the prepareCards() method, which takes care of loading all the cards in the deck with a double for loop that iterate the elements of the array. The toString() method instead creates and returns the representative string of the deck of cards.

Finally, the main class (the one with the main() method) is the following:

```
public class Exercise10V {
    public static void main(String args[]) {
        CardsDeck cardsDeck = new CardsDeck();
        System.out.println(cardsDeck);
    }
}
```

Where with a double loop we printed the forty cards in the deck.

The output is as follows (unfortunately the page is too narrow with respect to each line to be printed, so the output is a little staggered):

```
Ace of Cups, Two of Cups, Three of Cups, Four of Cups, Five of Cups, Six
  of Cups, Seven of Cups, Eight of Cups, Nine of Cups, Ten of Cups
Ace of Sticks, Two of Sticks, Three of Sticks, Four of Sticks, Five of
  Sticks, Six of Sticks, Seven of Sticks, Eight of Sticks, Nine of
  Sticks, Ten of Sticks
Ace of Coins, Two of Coins, Three of Coins, Four of Coins, Five of Coins,
  Six of Coins, Seven of Coins, Eight of Coins, Nine of Coins, Ten of
  Coins
Ace of Swords, Two of Swords, Three of Swords, Four of Swords, Five of
  Swords, Six of Swords, Seven of Swords, Eight of Swords, Nine of
  Swords, Ten of Swords
```

> **Also note that the formatting is far from perfect. We will try to solve the problem in the next exercise.**

385

Solution 11.z)

The formatting problems are:

1. an unneeded comma and space is printed at the end of each line.

2. At the end of the string, two unnecessary letters are.

Let's fix the two problems with two checks (in bold in the underlying code) within the two loops of the method. First we use a ternary operator that adds a comma and a space only if we are not at the last iteration on the numbers, and then with a `if` we add a "newline" only if we are not at the last iteration of the seeds:

```java
//...
    public String toString() {
        int cardsLength = cards.length;
        String string = "";
        for (int i = 0; i < cardsLength; i++) {
            int cardsRowLength = cards[i].length;
            for (int j = 0; j < cardsRowLength; j++) {
                string += cards[i][j] + (j != (cardsRowLength-1) ? ", " : "");
            }
            if (i != (cardsLength-1)) {
                string += "\n";
            }
        }
        return string;
    }
//...
```

Chapter 12
Exercises

Generic Types

Generic types are a topic that can become very complex. Among these exercises, and in particular among those with multiple answers that support the Oracle Java certifications, you will find some really complicated ones, because to answer correctly you will have to have an excellent preparation.

1. Generic types and type parameter are the same thing.

2. A main advantage of generics is that they allow you to avoid bugs like those caused by a `ClassCastException` at runtime, identifying them in the compilation phase.

3. The collections can also be used without specifying the type parameters. In this case we speak of raw type.

4. Inheritance ignores type parameters.

5. Wildcards are used when we can't specify type parameters.

6. If we create a generic type with type parameter <E>, we can use the same parameter also in the methods declared in the type.

7. In generic methods the type parameter is set as the input parameter of the method.

8. The following code:

```
public class MyGeneric <Foo extends Number> {
    private List<Foo> list;
    public MyGeneric () {
        list = new ArrayList<Foo>();
    }
    public void add(Foo pippo) {
        list.add(pippo);
    }
    public void remove(int i) {
        list.remove(i);
    }
    public Foo get(int i) {
        return list.get(i);
    }
    public void copy(ArrayList<?> al) {
        Iterator<?> i = al.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            add(o);
        }
    }
}
```

compile without errors.

9. The following code:

```
public <N extends Number> void print(List<N> list) {
    for (Iterator<N> i = list.iterator();
        i.hasNext( ); ) {
        System.out.println(i.next());
    }
}
```

compile without errors.

10. To resolve wildcard capture, we nned to use a helper method defined in the standard library.

11. The following code:

```
List<String> strings = new ArrayList<String>();
strings.add(new Character('A'));
```

compile without errors.

12. The following code:

```
List<int> ints = new ArrayList<int>();
```

compile without errors.

388

13. The following code:

```
List<int> ints = new ArrayList<Integer>();
```

compile without errors.

14. The following code:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
```

compile without errors.

15. The following code:

```
List ints = new ArrayList<Integer>();
```

compile without errors.

Exercise 12.b)

Consider the sources created with the exercises of Chapter 6 and then redefined in Chapter 9 and 10. Replace the `coins` array defined in the `Purse` class with an `ArrayList` of `Coin` objects. If necessary, modify the other classes so that everything works as before

> **Tip: there may be useful methods in the `ArrayList` class documentation.**

Exercise 12.c)

Create an abstract `Fruit` class that defines an encapsulated `weight` variable. For our program a `fruit` object without `weight` makes no sense. Create the subclasses `Apple`, `Peach` and `Orange`. Create a generic class `Basket` that abstracts the concept of fruit basket. This defines an array list of fruits. He must also expose a `getWeight()` method which returns the total weight of the contents of the basket; an `add()` method to add one fruit at a time, which raises a custom exception in case the addition of the fruit you want to add makes it exceed the maximum limit of 5 kilos in weight. Each basket must have only one type of fruit at a time. Implement a solution with generics. Create a test class to verify the actual operation of the program.

Exercise 12.d)

Which of the following statements compile without errors (choose all the correct statements):

1. `HashMap hm = new HashMap();`

2. `HashMap<> hm = new HashMap<>();`

3. `HashMap<Integer, String> hm = new HashMap ()<Integer, String>;`

4. `HashMap<Double> hm = new HashMap<Double>();`

5. `HashMap<Double> hm = new HashMap<>();`

Exercise 12.e)

Which of the following statements compile without errors (choose all the correct statements):

1. `ArrayList al = new ArrayList<Integer>();`

2. `ArrayList<Integer> al = new ArrayList<>();`

3. `ArrayList<String, String> al = new ArrayList<String, String>();`

4. `ArrayList al = new ArrayList<>();`

5. `ArrayList al = new ArrayList<int>();`

Exercise 12.f)

Which of the following statements compile without errors (choose all the correct statements):

1. `List al = new ArrayList<HashMap<String, String>>();`

2. `Map<ArrayList<String>> hm = new ArrayList<>();`

3. `List<String, String> al = new ArrayList<HashMap<String, String>>();`

4. `List<Map<List<List<Integer>>, String>> al = new ArrayList<>();`

5. `List<Map<String, String>> al = new ArrayList<HashMap<String, String>>();`

Exercise 12.g)

What is the output of the following code?

```
import java.util.*;

public class Exercise12G {
```

```
    public static void main(String args[]) {
        List list = new ArrayList();
        list.add("1");
        list.add('2');
        list.add(3);
        Iterator iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
        }
    }
}
```

1. 123

2. 321

3. None, the program cannot be compiled because it is not possible to add different elements to the collection.

4. "1" '2'3

5. None, the program cannot be compiled because it is not possible to retrieve an Iterator from a heterogeneous collection.

6. The execution will be stopped when we try to print the second value of the collection.

Exercise 12.h)

What is the output of the following code?

```
import java.util.*;

public class Exercise12H {

    public static void main(String args[]) {
        List<Object> list = new ArrayList<>();
        list.add("1");
        list.add('2');
        list.add(3);
        Iterator<Object> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
        }
    }
}
```

1. 123

2. "1"'2'3

3. None, the program cannot be compiled because the collection only admits objects of type `Object`.

4. None, the program cannot be compiled because it is not possible to retrieve an `Iterator` from a heterogeneous collection.

5. The execution will be stopped when we try to print the second value of the collection.

Exercise 12.i)

What is the output of the following code?

```java
import java.util.*;

public class Exercise12I {

    public static void main(String args[]) {
        List<String> list = new ArrayList<>();
        list.add("1");
        list.add(null);
        list.add('3');
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
        }
    }
}
```

1. None, the program cannot be compiled because the collection does not allow `null` values.

2. None, the program cannot be compiled because the collection does not allow primitive values.

3. The execution will be stopped when you try to add `null` to the collection.

4. The execution will be stopped when you try to add '3' to the collection.

5. `1null3`

Exercise 12.l)

What is the output of the following code?

```java
import java.util.*;
```

```
public class Exercise12L {

    public static void main(String args[]) {
        Map<Integer, Integer> map = new HashMap<>(1);
        map.put(14, 12);
        map.put('a','b');
        map.put(07,0x0ABC);
    }
}
```

1. The code does not compile because it is not possible to add three key-value pairs to a collection of size 1.

2. The code does not compile because the values 07 and 0x0ABC are not integer values.

3. The code does not compile because the values 'a' and 'b' are not integer values.

4. None of the above.

Exercise 12.m)

Given the following statement:

```
Map<String, Float> map = new Hashmap<>(1);
```

Which of the following statements are valid?

1. map.add("string", 1.1F);

2. map.add("string", 1.1D);

3. map.add("string", 1);

4. None of the above.

Exercise 12.n)

Given the following class:

```
public class Exercise12N/*INSERT CODE 1*/ {

    public static void main(String[] args) {
        Exercise12N<String> g = new Exercise12N/*INSERT CODE 2*/();
        Exercise12N<Object> g2 = new Exercise12N/*INSERT CODE 3*/();
    }
}
```

393

If we wanted to insert the three missing instructions for the compilation to be successful, which of these options will be valid?

1. Code 1: <Object>; Code 2: <Object>; Code 3: <Object>.

2. Code 1: <>; Code 2: <>; Code 3: <>.

3. Code 1: no code; Code 2: <>; Code 3: <>.

4. Code 1: <T extends Object>; Code 2: <String>; Code 3: <Object>.

5. Code 1: <? extends Object>; Code 2: <String>; Code 3: <Object>.

6. Code 1: <T>; Code 2: <String>; Code 3: <>.

7. Code 1: <T>; Code 2: <T>; Code 3: <Object>.

Exercise 12.o)

Given the following class:

```java
import java.util.*;

public class Exercise12O {

    public static int getSize(List/*INSERT CODE HERE*/ list) {
        return list.size();
    }

    public static void main(String args[]) {
        System.out.println(getSize(new ArrayList<Integer>()));
        System.out.println(getSize(
          new ArrayList<HashMap<String, List<String>>>()));
    }
}
```

Which of the following generic parameters could replace the comment /*INSERT CODE HERE*/ to ensure that the code compiles correctly, and allows you to print the size of the list passed as input, whatever the type of list:

1. <Object>

2. <>

3. <?>

4. <T extends Object>

5. `<? extends Object>`

6. `<T>`

7. No code, the file can be compiled as is.

Exercise 12.p)

Given the following class:

```java
public class Exercise12P<T> {

    public static T[] getValues(T t) {
        return t.values();
    }

    private enum T {
        T1, T2, T3;
    }
}
```

Which of the following statements are correct?

1. The code does not compile because it is not possible to call the nested enumeration `T` as the type parameter.

2. The code compiles correctly.

3. The code compiles correctly, but shows a warning.

4. The code does not compile because it is not possible to declare an enumeration as a type parameter.

5. The code does not compile because a static method cannot access a non-static enumeration.

6. None of the above.

Exercise 12.q)

Given the following class:

```java
public class Exercise12Q/*INSERT CODE HERE*/ {

    public Integer max(N n1, N n2) {
        return Integer.max(n1.intValue(), n2.intValue());
    }
}
```

Which of the following generic parameters could replace the comment /*INSERT CODE HERE*/ to make the code compile correctly:

1. `<Object>`

2. `<Number>`

3. `<?>`

4. `<T>`

5. `<N>`

6. `<N extends Number>`

7. `<T>`

8. No code, the file can be compiled as is.

Exercise 12.r)

Given the following class:

```java
import java.util.List;
import java.util.ArrayList;

public class Exercise12R<T> {

    public static void main(String args[]) {
        Exercise12R<Void> e = new Exercise12R<>();
        ArrayList<Integer> a = new ArrayList<>();
        a.add(2);
        a.add(6);
        a.add(10);
        a.add(24);
        a.add(17);
        System.out.println("The sum of the list elements is " +
            e.sumElements(a));
    }
}
```

Create a generic method (see section 12.3.3) which:

■ take as input a generic parameter L defined by the method, which represents a list of integers.

■ Have an integer as the return type, which is the sum of all the elements in the integer list.

396

> **Note that we have used the** `Void` **type as generic type for the** `Exercise12R`
> **instance, since the** `Exercise12R` **class does not use the parameters.**

Exercise 12.s)

Given the following hierarchy:

```
public interface Technology {
    void print();
}

public class Inkjet implements Technology {
    @Override
    public void print(){
        System.out.println("Inkjet print");
    }
}

public class Laser implements Technology {
    @Override
    public void print(){
        System.out.println("Laser print");
    }
}
```

Create a generic `Printer` class parameterized with a technology, which in turn declares a `print()` method, which however delegates to its own technology the actual message to be printed. Also create a class that tests the actual functioning of our code

Exercise 12.t)

Given the following enumeration:

```
public enum Operator {

    SUM("+"), SUBTRACTION("-"), MULTIPLICATION("X"), DIVISION(":");

    String sign;

    Operator(String sign) {
        this.sign = sign;
    }
```

397

```
    public String toString() {
        return sign;
    }
}
```

and the class:

```
public class OperationTest {
    public static void main(String args[]) {
        Operation<Integer, Operator> operation =
            new Operation<>(1, Operator.SUM, 1);
        operation.print();
    }
}
```

create the Operation class, with a print() method that simply prints:

```
1 + 1
```

Exercise 12.u)

Given the following class:

```
import java.util.List;
import java.util.Iterator;

public class Exercise12U {
    public static <T extends Number> void loop(List<T> list) {
        for (Iterator<T> i = list.iterator(); i.hasNext( ); ) {
            System.out.println(i.next());
        }
    }
}
```

Which of the following statements about the loop() method are true?

1. It is a method that uses a lower bound wildcard.

2. It is a method that uses a bounded parameter.

3. It is a method that uses an upper bounded wildcard.

4. It is a generic method.

5. It's a method that uses a wildcard capture.

6. It is a helper method.

7. It is a method that has a covariant parameter.

398

Exercise 12.v)

Given the following class:

```
import java.util.List;

public class Exercise12V {
    protected final void update(List<?> list) {
        list.add(list.get(0));
    }
}
```

Which of the following statements regarding the modification update() are true?

1. It is a method that does not compile.

2. It is a generic method.

3. It is a helper method.

4. It needs a helper method.

Exercise 12.z)

Given the following class:

```
public class Exercise12Z {
    public static <E extends Exception> void printException(E e) {
        System.out.println(e.getMessage());
    }
    public static void main(String[] args) {
        /*INSERT CODE HERE*/
        Exercise12Z.printException(new Throwable("Exception"));
    }
}
```

Which of the following statements could replace the comment /*INSERT CODE HERE*/ to ensure that the code compiles correctly:

1. `Exercise12Z.<Exception>printException(new Exception("Exception"));`

2. `Exercise12Z.printException(new ClassCastException("ClassCastException"));`

3. `Exercise12Z.printException(new RuntimeException("RuntimeException"));`

4. `Exercise12Z.printException(new Throwable("Exception"));`

Chapter 12
Exercise Solutions

Generic Types

1. **False**, see. section 12.1.

2. **True**.

3. **True**.

4. **True**.

5. **True**.

6. **True**.

7. **False**.

8. **False**, we will get the following error:

```
(Foo) in MyGeneric<Foo> cannot be applied to
(java.lang.Object)
add(o);
^
```

9. **True**.

10. **False**, the helper method must be created by hand.

11. **False**.

12. **False**.

13. **False**.

14. **True**.

15. **True**.

Solution 12.b)

The Purse class code will change a lot:

```java
import java.util.ArrayList;
import java.util.List;

/**
 * Abstracts the concept of a purse that can hold a limited number of
 * coins.
 *
 * @author Claudio De Sio Cesari
 */
public class Purse {

    /**
     * Constant that represents the maximum number of coins that this
     * purse can hold
     */
    private static final int DIMENSION = 10;

    /**
     * An array list that contains a limited number of coins.
     */
    private final List<Coin> coins = new ArrayList<>(DIMENSION);

    /**
     * Create a purse containing coins whose values are specified by the
     * varargs values. lf the number of elements of the varargs values is
     * greater than the maximum number of coins that the current purse can
     * hold, then it is handled an exception
     *
     * @param values un varargs di values di coins.
     */
    public Purse(Value... values) {

        assert coins.size() <= DIMENSION;
```

```
      try {
          int numberOfCoins = values.length;
          for (int i = 0; i < numberOfCoins; i++) {
              if (i >= DIMENSION) {
                  throw new FullPurseException (
                      "Only the first 10 coins have been inserted!");
              }
              coins.add(new Coin(values[i]));
          }
      } catch (FullPurseException exc) {
          System.out.println(exc.getMessage());
      } catch (NullPointerException exc) {
          System.out.println("The purse has been created empty");
      }
      assert coins.size() <= DIMENSION;
  }

  /**
   * Adds a coin to the purse. If this is full the coin will not
   * be added and a significant error will be printed.
   *
   * @param coin
   *           the coin to add.
   * @throws FullPurseException
   */
  public void add(Coin coin) throws FullPurseException {
      try {
          System.out.println("Let's try adding one "+
              coin.getDescription());
      } catch (NullPointerException exc) {
          throw new NullCoinException();
      }
      int freeIndex = firstFreeIndex();
      if (freeIndex == -1) {
          throw new FullPurseException("Purse full! The coin "
              + coin.getDescription() +" has not been added!");
      } else {
          coins.set(freeIndex, coin);
          System.out.println(coin.getDescription() +" has been added");
      }
  }

  /**
   * Performs a withdrawal of the specified coin from the current coin purse.
   * In case the specified currency is not present, a significant error
   * will be printed and null will be returned.
   *
   * @param coin
   *           the coin to take.
```

403

```
 * @return
 *             the coin found.
 * @throws CoinNotFoundException
 *             if the coin is not found.
 */
public Coin withdraw(Coin coin) throws CoinNotFoundException {
    try {
        System.out.println("Let's try to get a " +
            coin.getDescription());
    } catch (NullPointerException exc) {
        throw new NullCoinException();
    }
    Coin foundCoin = null;
    int foundCoinIndex = foundCoinIndex(coin);
    if (foundCoinIndex == -1) {
        throw new CoinNotFoundException("Coin not found!");
    } else {
        foundCoin = coin;
        coins.set(foundCoinIndex, coin);
        System.out.println("One " + coin.getDescription() + " withdrawn");
    }
    return foundCoin;
}

/**
 * Print the contents of the purse.
 */
public void state() {
    System.out.println("The purse contains:");
    for (Coin coin : coins) {
        if (coin == null) {
            break;
        }
        System.out.println("One " + coin.getDescription());
    }
}

/**
 * Retrieves the first free index in the coin array or -1 if the
 * coin purse is full.
 *
 * @return
 *             the first free index in the coin array or -1 if the
 *             coin purse is full.
 */
private int firstFreeIndex() {
    int index = coins.indexOf(null);
    return index;
}
```

404

```
    private int foundCoinIndex(Coin coin) {
        int foundCoinIndex = -1;
        final int size = coins.size();
        for (int i = 0; i < size; i++) {
            if (coins.get(i) == null) {
                continue;
            }
            Value coinInPurseValue = coins.get(i).getValue();
            Value value = coin.getValue();
            if (value == coinInPurseValue) {
                foundCoinIndex = i;
                break;
            }
        }
        return foundCoinIndex;
    }
}
```

The constructor now has different assertions, because the size of an ArrayList is measured differently from the size of an array, but it has kept its algorithmic logic and has only changed the way an element is added to the array list (add()).

The same logic also applies to the add() method, which has only changed the way an element with a specified index is added to the array list (set() method).

The firstFreeIndex() private method has been greatly simplified, thanks to the use of the indexOf() defined by ArrayList.

The reader should now also be able to find the differences in the withdraw() method and in the foundCoinIndex() method, since they are very similar to what we have seen for the methods add() and firstFreeIndex().

It is not necessary to modify any other class as the variable coins we have modified is encapsulated.

Solution 12.c)

The code of the Fruit class and its subclasses is shown below:

```
public abstract class Fruit {

    private final int weight;

    public Fruit(int weight) {
        this.weight = weight;
    }

    public int getWeight() {
        return weight;
    }
}
```

```
public class Apple extends Fruit{
    public Apple(int weight) {
        super(weight);
    }
}

public class Peach extends Fruit {
    public Peach(int weight) {
        super(weight);
    }
}

public class Orange extends Fruit {
    public Orange(int weight) {
        super(weight);
    }
}
```

The class representing the exception could be the following:

```
public class WeightException extends Exception {
    public WeightException(String message) {
        super(message);
    }
}
```

The Basket class instead, could be coded as the following:

```
import java.util.ArrayList;

public class Basket<F extends Fruit> {

    private ArrayList<F> fruit;

    public Basket() {
        fruit = new ArrayList<>();
    }

    public ArrayList<F> getFruit() {
        return fruit;
    }

    public void addFruit(F fruitItem) throws WeightException {
        final int newWeight = getWeight() + fruitItem.getWeight();
        if (newWeight > 2000) {
            throw new WeightException("Too heavy: " + newWeight + " grams!");
        }
        fruit.add(fruitItem);
        System.out.println(fruitItem.getWeight() + " gramms of "+
            fruitItem.getClass().getName() +" added to basket");
    }
```

406

```
    public int getWeight() {
        int weight = 0;
        for (F fruit : fruit) {
            weight += fruit.getWeight();
        }
        return weight;
    }
}
```

Note that with the parameter type we can use any subclass of Fruit. The F type is then used in class methods and will be replaced during execution with a subclass of Fruit.

Finally, we can code a simple test class in the following way:

```
public class FruitTest {
    public static void main(String args[]) {
        Basket<Apple> appleBasket = new Basket<>();
        for (int i = 0; i <= 20; i++) {
            try {
                Apple apple = new Apple(100);
                appleBasket.addFruit(apple);
            } catch (WeightException exc) {
                System.out.println(exc.getMessage());
            }
        }
    }
}
```

The output of the FruitTest class execution follows:

```
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
100 gramms of Apple added to basket
Too heavy: 2100 grams!
```

Solution 12.d)

The correct answer is number 1. The number 2 is incorrect because the diamond operator is used on the instance and not on the reference. The number 3 is incorrect for the incorrect order of the round brackets and the generic parameters of the instance (if we reverse the order of the parameters, we would get a correct statement). The last two are incorrect because a HashMap always has two parameters.

Solution 12.e)

The correct answers are the number 1, the number 2 and the number 4. The number 4, in particular, is a special case because it is a raw type that uses the diamond operator, which in this particular case is practically optional. The number 3 is incorrect because an ArrayList always has a single generic parameter.

Solution 12.f)

The correct answers are the number 1 and the number 4. The number 2 is incorrect because a map always has two parameters. The number 3 does not compile because there is no coincidence between the reference parameters (two strings, and this is already an error because a list always has only one parameter) and the instance (a HashMap). The number 5 is incorrect because the generic parameters between reference and instance do not match. Indeed, Map does not coincide with HashMap.
Here is the output using JShell:

```
jshell> List<Map<String, String>> al =
   new ArrayList<HashMap<String, String>>() ;
|   Error:
|   incompatible types: java.util.ArrayList<
   java.util.HashMap<java.lang.String,java.lang.String>> cannot be
   converted to java.util.List<java.util.Map<
   java.lang.String,java.lang.String>>
|   List<Map<String, String>> al =
   new ArrayList<HashMap<String, String>>();
|   ^------------------------------------^
```

Solution 12.g)

The correct answer is 1. In fact, not having specified the generic parameter (we speak of raw type in these cases), it will be possible to add elements of any kind to the collection. Note that adding primitive parameters, these are automatically promoted to the relative wrapper objects

(in this case '2' is boxed in a `Character`, while 3 in a `Integer`). The output of the execution follows:

```
123
```

Solution 12.h)

The correct answer is. It follows the runtime output:

```
123
```

Solution 12.i)

The correct answer is 2. It is not possible to add a primitive type (or other complex types other than `String`) to a collection that is parameterized with a `String` type. In fact, the compilation output is the following:

```
error: no suitable method found for add(char)
        list.add('3');
            ^
    method Collection.add(String) is not applicable
      (argument mismatch; char cannot be converted to String)
    method List.add(String) is not applicable
      (argument mismatch; char cannot be converted to String)
Note: Some messages have been simplified; recompile with
  -Xdiags:verbose to get full output
1 error
```

Solution 12.l)

The only correct statement is number 3.

The statement number 1 is not valid because the collections are resizable by default, and the value 1 passed to the `HashMap` constructor, represents only the initial size of the collection.

Also the number 2 statement is incorrect. In fact, the values 07 and 0x0ABC are integer values and therefore with the autoboxing they are correctly "wrapped" inside `Integer` objects.

Instead the values 'a' and 'b' of which we speak in the statement 3, are characters and are "wrapped" in `Character` objects.

The output of the compilation follows:

```
error: incompatible types: char cannot be
  converted to Integer
        map.put('a','b');
```

```
              ^
Note: Some messages have been simplified; recompile with
  -Xdiags:verbose to get full output
1 error
```

Solution 12.m)

The answer is the last one, since to add key-value pairs to a map, the `put()` method must be used, not the `add()` method (which is a method declared in the lists).

Solution 12.n)

The correct options are 4 and 6.

Option number 1 is incorrect because you cannot assign a reference with generic type `String`, to an instance with generic type `Object`. In fact, by compiling the file we get the following output:

```
error: incompatible types: Exercise12N<Object>
  cannot be converted to Exercise12N<String>
    Exercise12N<String> g = new Exercise12N<Object>();
                                ^
  where Object is a type-variable:
    Object extends java.lang.Object declared in class Exercise12N
1 error
```

The option number 2 will fail at the first line, because it is not possible to use a diamond operator as a generic type of a class. The number 3 will also fail to compile because we have not declared a generic type for the class. Following is the output that reports the errors deriving from the compilation process:

```
Exercise12N.java:2: error: type Exercise12N does not take parameters
    Exercise12N<String> g = new Exercise12N<>();
              ^
Exercise12N.java:3: error: type Exercise12N does not take parameters
    Exercise12N<Object> g2 = new Exercise12N<>();
              ^
Exercise12N.java:2: error: cannot infer type arguments for
  Exercise12N
    Exercise12N<String> g = new Exercise12N<>();
                                          ^
  reason: cannot use '<>' with non-generic class Exercise12N
Exercise12N.java:3: error: cannot infer type arguments for
  Exercise12N
    Exercise12N<Object> g2 = new Exercise12N<>();
                                           ^
  reason: cannot use '<>' with non-generic class Exercise12N
4 errors
```

410

The number 5 is also incorrect. In fact, wildcards are used with method parameters. The compilation output follows:

```
Exercise12N.java:1: error: <identifier> expected
public class Exercise12N<? extends Object> {
                              ^
1 error
```

Finally, the number 7 produces the following compilation output:

```
Exercise12N.java:2: error: incompatible types: Exercise12N<T>
  cannot be converted to Exercise12N<String>
    Exercise12N<String> g = new Exercise12N<T>();
                            ^
  where T is a type-variable:
    T extends Object declared in class Exercise12N
1 error
```

where we are warned that `Exercise12N<T>`, cannot be converted to `Exercise12N<String>`.

Solution 12.o)

The correct options are 3 and 5, which are essentially equivalent, but also 7 where we use a raw type. In fact, by compiling using option 7 we will get a warning:

```
Exercise12O.java:4: warning: [rawtypes] found raw type: List
    public static int getSize(List/*INSERT CODE HERE*/ list) {
                              ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
1 warning
```

Option 1 is incorrect, because the `Object` parameter does not allow other parameter types outside `Object`. In fact, here is the output:

```
Exercise12O.java:9: error: incompatible types: ArrayList<Integer>
  cannot be converted to List<Object>
        System.out.println(getSize(new ArrayList<Integer>()));
                                   ^
Exercise12O.java:10: error: incompatible types:
  ArrayList<HashMap<String,List<String>>> cannot be converted to
  List<Object>
        System.out.println(getSize(new ArrayList<
  HashMap<String, List<String>>>()));
                                   ^
Note: Some messages have been simplified; recompile with
  -Xdiags:verbose to get full output
2 errors
```

Option 2 makes no sense (incorrect syntax). In fact, here is the output:

```
Exercise12O.java:4: error: illegal start of type
    public static int getSize(List<> list) {
                                   ^
1 error
```

Option 3 and option 6 are both wrong for the same reason. Type T is not defined anywhere. Follow the output with option 3:

```
Exercise12O.java:4: error: cannot find symbol
    public static int getSize(List<T> list) {
                                   ^
  symbol:   class T
  location: class Exercise12O
1 error
```

which is practically identical to the one where option 6 is used:

```
Exercise12O.java:4: error: > expected
    public static int getSize(List<T extends Object> list) {
                                    ^
1 error
```

Solution 12.p)

The correct statement is the number 3. In fact the output is the following:

```
Exercise12P.java:4: warning: [static] static method should be
  qualified by type name, T, instead of by an expression
        return t.values();
                ^
1 warning
```

That is, the compiler is warning us that the values() method is static and therefore should be called using the name of the enumeration, and not on one of its elements.
For the rest of the answers it is easy to understand why they are not correct. In particular for the number 1 the letter T is the identifier of the enumeration, therefore the question is put in a completely wrong way.

Solution 12.q)

The correct option is the number 6. Where there is no Number type bounded parameter, we will get a compilation error like the following:

```
Exercise12Q.java:4: error: cannot find symbol
        return Integer.max(n1.intValue(), n2.intValue());
                             ^
  symbol:   method intValue()
  location: variable n1 of type N
  where N is a type-variable:
    N extends Object declared in class Exercise12Q
Exercise12Q.java:4: error: cannot find symbol
        return Integer.max(n1.intValue(), n2.intValue());
                                            ^
  symbol:   method intValue()
  location: variable n2 of type N
  where N is a type-variable:
    N extends Object declared in class Exercise12Q
2 errors
```

Solution 12.r)

The solution could be the following (the requested method is highlighted in bold):

```java
import java.util.List;
import java.util.ArrayList;

public class Exercise12R<T> {

    private <L extends List<Integer>> Integer sumElements(L list) {
        int size = list.size();
        int result = 0;
        for (int i = 0; i < size; i++) {
            Integer item = list.get(i);
            result += item;
        }
        return result;
    }

    public static void main(String args[]) {
        Exercise12R<Void> e = new Exercise12R<>();
        ArrayList<Integer> a = new ArrayList<>();
        a.add(2);
        a.add(6);
        a.add(10);
        a.add(24);
        a.add(17);
        System.out.println("The sum of the list elements is "
            + e.sumElements(a));
    }
}
```

Solution 12.s)

A possible solution is represented by the following code, which declares a classic bounded parameter, and, as required, declares the `print()` method which delegates its real operation to the implementation of the `print()` method defined by the `technology` variable.

```java
public class Printer<T extends Technology> {

    private T technology;

    public Printer(T technology){
        this.technology = technology;
    }

    public void print(){
        technology.print();
    }
}
```

The actual operation can be tested by the following class of test:

```java
public class PrinterTest {
    public static void main(String args[]) {
        Printer<Laser> printer = new Printer<>(new Laser());
        printer.print();
        Printer<Inkjet> printer2 = new Printer<>(new Inkjet());
        printer2.print();
    }
}
```

which produces the following output:

```
Laser print
Inkjet print
```

Solution 12.t)

The `Operation` class could be the following:

```java
public class Operation<Integer, O extends Operator> {

    private Integer operand1;
    private O operator;
    private Integer operand2;

    public Operation(Integer operand1, O operator, Integer operand2) {
        this.operand1 = operand1;
        this.operator = operator;
        this.operand2 = operand2;
    }
```

```
    public void print(){
        System.out.println(operand1  +  " " + operator +  " " + operand2);
    }
}
```

Solution 12.u)

The correct statements are 3 and 4.

Solution 12.v)

The correct statements are 1 and 4. In particular we should create a helper method like the following in bold:

```
import java.util.List;

public class Solution11V {
    protected final void modifica(List<?> list) {
        helperMethod(list);
    }

    private <T> void helperMethod(List<T> list) {
        list.add(list.get(0));
    }
}
```

Solution 12.z)

The correct statements are the first 3, only the use of statement 4 would cause an error in compilation, follows the output:

```
Exercise12Z.java:6: error: method printException in class
  Exercise12Z cannot be applied to given types;
        /*INSERT CODE HERE*/Exercise12Z.printException(
  new Throwable("Exception"));
                                             ^
  required: E
  found: Throwable
  reason: inference variable E has incompatible bounds
    upper bounds: Exception
    lower bounds: Throwable
  where E is a type-variable:
    E extends Exception declared in method <E>printException(E)
1 error
```

Nel caso della risposta numero 1, abbiamo usato una sintassi che si utilizza raramente, e di cui abbiamo solo accennato alla fine del section 12.3.3 relativamente ad un costruttore.

Chapter 13
Exercises

The Indispensable Library: The `java.lang` Package

As with all exercises in this book, you can consult the documentation of the standard library to find solutions. This is especially true in the chapters like these dedicated to libraries.

Exercise 13.a) Autoboxing, autounboxing and `java.lang`, *True or False:*

1. The following code compiles without errors:
   ```
   char c = new String("Foo");
   ```
2. The following code compiles without errors:
   ```
   int c = new Integer(1) + 1 + new Character('a');
   ```
3. The overload rules do not change with the introduction of autoboxing and auto-unboxing.
4. `Integer` class instances are immutable, so their internal state cannot be changed once instantiated.
5. The `Runtime` class depends strictly on the operating system on which it runs.
6. The `Class` class allows you to read members of a class (but also superclasses and other information) simply starting from the class name thanks to the `forName()` method.

7. The `Class` class allows you to instantiate objects of a class knowing only the name.

8. It is possible from Java version 1.4 to sum a primitive type and an object of its wrapper class, as in the following example:

```
Integer a = new Integer(30);
int b = 1;
int c = a + b;
```

9. Object cloning requires a call to `Object`'s `clone()` method.

10. The `Math` class cannot be instantiated because it is declared abstract.

Exercise 13.b)

Let's consider the `Purse` class refined in the exercises of Chapter 10, and focus on the private method `foundCoinIndex()`, which we defined in the following way:

```
private int foundCoinIndex(Coin coin) {
    int foundCoinIndex = -1;
    final int size = coins.size();
    for (int i = 0; i < size; i++) {
        if (coins.get(i) == null) {
            continue;
        }
        Value coinInPurseValue = coins.get(i).getValue();
        Value value = coin.getValue();
        if (value == coinInPurseValue) {
            foundCoinIndex = i;
            break;
        }
    }
    return foundCoinIndex;
}
```

Create an `equals()` method in the `Coin` class, so as to simplify the search in this method.

Exercise 13.c)

Create a static block (see section 6.8.3) in the `Coin` class, which we have refined in the exercises of Chapter 10, and make it print any sentence. How can reflection be used to make the block execute?

Exercise 13.d)

Create an `InteractiveReflection` class that contains a `main()` method which prints the methods of a class specified at runtime through the use of a `Scanner` class (already encountered

several times previously, as in Exercise 4.m in the `InteractiveProgram` class). It must be possible to specify a class, press the **Enter** key and the program must print the signatures of all the methods of the specified class.

> **This program may not run correctly with EJE, it is recommended to run it from the command line.**

Exercise 13.e)

Create an `InteractiveCompiler` class that contains a `main()` method which compiles the files specified at runtime through the use of a `Scanner` class. It must be possible to specify a class, press the **Enter** key and the program must compile the specified class.

> **This program should be run on the command line or via EJE, but not via Eclipse or Netbeans. In fact, these IDEs automatically compile the files and therefore you would not be able to understand well if the file is correctly compiled by our program or by the IDE.**

Exercise 13.f)

Create a class that, given an array of integers, sorts them from highest to lowest.

Exercise 13.g)

Create a `WrapperComparable` class that has the following characteristics:

- It has an `Integer` encapsulated constant that must always have a value.
- Have a `toString()` method that returns the integer.
- Define a sorting method that goes from the highest to the lowest encapsulated integer.

Finally create a class that tests that the ordering works as expected.

Exercise 13.h)

What is the output of the following class?

```
public class Exercise13H {
    public static void main(String args[]) {
        String string1 = "Claudio";
        String string2 = new String(stringa1);
        System.out.println(string2 == string1);
        System.out.println(string2.equals(string1));
        System.out.println("Claudio".equals(string1));
        System.out.println("Claudio" == string1);
        System.out.println("Claudio" == string2);
    }
}
```

Exercise 13.i)

Which of the following statements are correct?

■■ An immutable object cannot be modified.

■■ Strings are immutable objects.

■■ Instances of wrapper classes are immutable objects.

■■ An immutable object can be pointed to by multiple references.

■■ An immutable object does not allow you to change its internal state.

■■ An immutable object does not allow you to change its external status.

Exercise 13.l)

Which of the following statements are correct with respect of compact strings topic?

■■ You can specify which strings should be made compact.

■■ A non-compact string uses 16 bits.

■■ A compact string uses only 8 bits.

■■ A program, to make all the strings compact, must be compiled using the option -XX:-CompactStrings.

■■ Using compact strings, program performance will always be increased by 50%.

Exercise 13.m)

What is the output of the following class?

420

```
public class Exercise13M {
    public static void main(String args[]) {
        String string = "*** Java ***";
        string.toUpperCase();
        string.trim();
        string.substring(3, 8);
        string.trim();
        string.concat(String.format("String = %n", string.length()));
        string += "!";
        System.out.println(string.length);
    }
}
```

1. 12

2. 13

3. 11

4. 10

5. 23

6. 22

7. 24

8. No output, a runtime exception will be thrown.

9. No output, the class cannot be compiled.

Exercise 13.n)

What is the output of the following class?

```
public class Exercise13N {
    public static void main(String args[]) {
        String string = "Java";
        string = string.concat(" ");
        string += 9;
        String result = "";
        if (string.intern() == "Java for Aliens") {
            result += "intern()";
        }
        if (string == "Java for Aliens") {
            result += "==";
        }
        if (string.equals("Java for Aliens")) {
```

```
                  result += "equals()";
            }
            System.out.println(result);
    }
}
```

1. Java for Aliens

2. intern()

3. intern()==

4. intern()equals()

5. null

6. intern()==equals()

7. An empty string.

8. No output, a runtime exception will be thrown.

9. No output, the class cannot be compiled.

Exercise 13.o)

What is the output of the following class?

```
public class Exercise13O {
    public static void main(String args[]) {
        String string1 = "123789";
        String string2 = string1.concat(System.lineSeparator());
        char [] array1 = string2.toCharArray();
        char [] array2 = {'4', '5', '6'};
        System.arraycopy(array2, 0, array1, 3, 3);
        System.out.println(array1);
        System.exit(0);
    }
}
```

Exercise 13.p)

Write a class that represents a text of the RTF type (Rich Text Format), that is a text to which it is possible to modify for example the type of character, the background color, the line spacing, the underlining and so on (you choose what must define). Make this class clonable and create a test program that verifies the actual functioning.

422

Exercise 13.q)

Starting from exercise 6.z, create a class with the `main()` method that reads username and password as system properties. Test the class using the right command line options (specify them).

Exercise 13.r)

Which of the following statements about garbage collection are correct?

1. The JVM only implements two algorithms to deallocate memory: G1GC and ParallelGC.

2. With ParallelGC the interventions on memory are more numerous and intense, and this makes the algorithm less efficient compared to G1GC.

3. It was possible to use G1GC already from Java version 7.

4. The "finalization" consists in eliminating the objects no longer used by the application.

5. The "finalization" can be invoked by calling the `runFinalization()` method of the `Object` class.

6. To use ParallelGC it is necessary to run the application specifying the option `-XX: + UseParallelGC`.

Exercise 13.s)

We mentioned that up to Java 8, the `setAccessible()` method called on a `Field` object (which we remember abstracts the concept of variable) or a `Method` object (which abstracts the concept of method), allowed reflection to access private members of a class, effectively violating encapsulation. With the introduction of modules in Java 9, this is no longer possible.
So, given the following class:

```
public class ClassWithPrivateMembers {
    private String privateVariable = "This variable is private and cannot " +
        be touched!";
    private String privateMethod() {
        return "This method is private and cannot be touched!";
    }
}
```

By consulting the official documentation, create another class that with reflection:

- try to modify the value of the variable `privateVariable` variable;

- try to invoke the `privateMethod()` method.

423

Given the following class:

```
public class Exercise13T {
    public static void main(String args[]) {
        int radius = 7;
        /*INSERT CODE HERE*/
        System.out.println("The area of the circumference with radius 7 is "
            + area);
    }
}
```

define the line that must replace the comment /*INSERT CODE HERE*/ to get the correct area calculation.

> **Remember that the area of a circle is calculated as:**
>
> **pi for squared radius, or if we call the radius `r`, and `A` the area of the circle, and we denote the pi number with ⅂⅂, we will have**
>
> `A = ⅂⅂r²`
>
> **Consult the official documentation of the `Math` class before writing the code.**

What is the output of the following class?

```
public class Exercise13U {

    public static void main(String args[]) {
        double e = Math.E;
        Math.floor(e);
        boolean b = check(e, 2.0);
        System.out.println(b);
    }

    public static Boolean check(Double a, Double b) {
        Boolean equals = null;
        if (a.equals(b)) {
            equals = true;
        }
        return equals;
    }
}
```

424

1. `true`

2. `false`

3. `null`

4. `2.0`

5. `2.718281828459045`

6. No output, a runtime exception will be thrown.

7. No output, the class cannot be compiled.

Exercise 13.v)

What is the output of the following class?

```
public class Exercise13V {
    public static void main(String args[]) {
        Exercise13V e = new Exercise13V();
        e.method(128);
    }

    public void method(Integer number) {
        System.out.println("Integer " + number);
    }
    public void method(long number) {
        System.out.println("long " + number);
    }
    public void method(byte number) {
        System.out.println("byte " + number);
    }
    public void method(Byte number) {
        System.out.println("Byte " + number);
    }
    public void method(short number) {
        System.out.println("short " + number);
    }
    public void method(Double number) {
        System.out.println("Double " + number);
    }
    public void method(double number) {
        System.out.println("double " + number);
    }
}
```

Exercise 13.z)

Create a program that simulates the game known as "stone-paper-scissors". The game instructions can be found at this link: **https://en.wikipedia.org/wiki/Rock-paper-scissors**.
Once the program has been run, the user must specify whether to choose stone, paper or scissors, and at the same time the program must make its choice (randomly).

> **Try to use all the concepts learned so far to create this application (including enumerations). Remember that it is difficult to make choices, but with the methods of analysis that we have seen before we can proceed with more rigor and self-confidence.**

Chapter 13
Exercise Solutions

The Indispensable Library:
The `java.lang` Package

1. **False**.

2. **True**.

3. **True**.

4. **True**.

5. **True**.

6. **True**.

7. **True**.

8. **False**, from Version 1.5.

9. **False**.

10. **False**, it cannot be instantiated because it has a private constructor and is declared final.

Solution 13.b)

We have generated the equals() method code (together with the hashcode() method, which we report for completeness even if not necessary for the exercise) using Netbeans (we also used the java.util.Objects class that we will explain later in the book):

```java
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Coin other = (Coin) obj;
    if (this.value != other.value) {
        return false;
    }
    return true;
}

@Override
public int hashCode() {
    int hash = 7;
    hash = 97 * hash + Objects.hashCode(this.value);
    return hash;
}
```

So the foundCoinIndex() method can be simplified as follows:

```java
private int foundCoinIndex(Coin coin) {
    int foundCoinIndex = -1;
    final int size = coins.size();
    for (int i = 0; i < size; i++) {
        if (coins.get(i) == null) {
            break;
        }
        if (coins.get(i).equals(coin)) {
            foundCoinIndex = i;
            break;
        }
    }
    return foundCoinIndex;
}
```

Solution 13.c)

Reflection is not necessary. In fact, suppose we have defined the following static block in the Coin class:

```
     static {
         System.out.println("Loaded the Coin class with currency = " + CURRENCY);
     }
```

It would be enough to simply define a variable of the `Coin` class as follows:

```
  public class ReflectionTest {
      public static void main(String args[]) {
          Coin coin = new Coin(Value.FIFTY_CENTS);
      }
  }
```

The previous code would produce the following output:

```
Loaded the Coin class with currency = EURO
Created a coin value 50 cents of EURO
```

However, if we wanted to use reflection, the following code will not suffice:

```
         Class<Coin> classCoin = Coin.class;
         try {
             classCoin.newInstance();
         } catch (InstantiationException | IllegalAccessException ex) {
             ex.printStackTrace();
         }
```

which in fact will produce the following output:

```
java.lang.InstantiationException: Coin
      at java.lang.Class.newInstance(Class.java:418)
      at TestReflection.main(ReflectionTest.java:10)
Caused by: java.lang.NoSuchMethodException: Coin.<init>()
      at java.lang.Class.getConstructor0(Class.java:2971)
      at java.lang.Class.newInstance(Class.java:403)
      ... 1 more
```

This happens because with the `newInstance()` method the constructor without parameters is called, but it does not exist!
The solution is to call the correct constructor with the following code:

```
  try {
      Constructor<Coin> costruttore = classCoin.getConstructor(Value.class);
      costruttore.newInstance(Value.FIFTY_CENTS);
  } catch (NoSuchMethodException | SecurityException | InstantiationException |
          IllegalAccessException | IllegalArgumentException |
          InvocationTargetException ex) {
      ex.printStackTrace();
  }
```

429

Solution 13.d)

The `InteractiveReflection` class code could be as follows:

```java
import java.lang.reflect.Method;
import java.util.Scanner;

public class InteractiveReflection {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String string = "";
        System.out.println("Type the name of a class in the " +
            "current folder and type enter, or write \"end\" to end the " +
            "program");
        while (!(string = scanner.next()).equals("end")) {
            System.out.println("You typed " + string.toUpperCase() + "!");
            try {
                printMethods(string);
            } catch (ClassNotFoundException ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Program terminated!");
    }

    private static void printMethods(String string) throws
                                                ClassNotFoundException {
        Class objectClass = Class.forName(string);
        Method[] methods = objectClass.getDeclaredMethods();
        for (Method method : methods) {
            System.out.println(method);
        }
    }
}
```

Note that we have used the `getDeclaredMethods()` method instead of `getMethods()`, because the latter would have also printed the methods inherited from the superclasses (in case you specify the class `Coin` the methods of the `Object` superclass).
The output of the previous code is as follows:

```
Type the name of a class in the current folder and type enter, or write "end" to end the
 program
Coin
You typed COIN!
Loaded the Coin class with currency = EURO
public boolean Coin.equals(java.lang.Object)
public int Coin.hashCode()
public Value Coin.getValue()
```

```
public java.lang.String Coin.getDescription()
end
Program terminated!
```

Solution 13.e)

The code of the InteractiveCompiler class could be the following:

```java
import java.util.Scanner;

public class InteractiveCompiler {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String string = "";
        System.out.println("Type the name of a source Java file in the " +
          "current folder and type enter, or write \"end\" to end the program");
        while (!(string = scanner.next()).equals("end")) {
            System.out.println("You typed " + string.toUpperCase() + "!");
            try {
                buildClass(string);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Program terminated!");
    }

    private static void buildClass(String string) throws Exception {
        Runtime runtime = Runtime.getRuntime();
        Process process = runtime.exec("javac " + string);
        final int exitValue = process.waitFor();
        System.out.println(exitValue == 0 ? string + " compiled!" :
                            "Cannot compile " + string);
    }
}
```

Note that we could also use the Compiler API, with a simple code like the following:

```java
JavaCompilerTool compiler = ToolProvider.defaultJavaCompiler();
compiler.run(new FileInputStream("MyClass.java"),null, null);
```

Also note that we have captured the output code of the compilation process with the waitFor() method. If this is 0 then the file has been compiled correctly.
The output of our program will be:

```
java InteractiveCompiler
Type the name of a source Java file in the current folder and type enter, or write "end"
```

431

```
 to end the program
Coin
You typed COIN!
Cannot compile Coin
Coin.java
You typed COIN.JAVA!
Coin.java compiled!
```

From the command line we first tried to view all the ***.class** files with the command `dir * .class` (see appendix A), but there were none. Then we compiled the **InteractiveCompiler.java** class and checked that the `InteractiveCompiler` file was generated. Then we ran our program and in our interactive session we first tried to compile the **Coin** file, but this does not exist. Then we tried to compile the **Coin.java** file and we verified that the compilation was actually successful. Of course, the `Value` class and its anonymous classes have also been compiled.

Solution 13.f)

If we talk about an array of integers, since autoboxing-unboxing exists, we can also refer to an `Integer` array. Since this class is declared final, and therefore cannot be extended, it is unthinkable to create a subclass that implements `Comparable`. The only solution that remains is to create a `Comparator` class that reverses the integers:

```
import java.util.Comparator;

public class IntegerComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return -(o1.compareTo(o2));
    }
}
```

The test class could be the following:

```
import java.util.Arrays;

public class TestIntegerComparator {
    public static void main(String args[]) {
        Integer []array = {1942, 1947, 1971, 1984, 1976, 1974};
        Arrays.sort(array, new IntegerComparator());
        for (int integer : array) {
            System.out.println(integer);
        }
    }
}
```

whose output will be:

432

```
1984
1976
1974
1971
1947
1942
```

The code of the `WrapperComparable` class should be defined like this:

```java
public class WrapperComparable implements Comparable<WrapperComparable> {

    private Integer integer;

    public WrapperComparable(Integer integer) {
        if (integer == null) {
            integer = 0;
        }
        this.integer = integer;
    }

    public Integer getInteger() {
        return integer;
    }

    public void setInteger(Integer integer) {
        this.integer = integer;
    }

    @Override
    public int compareTo(WrapperComparable otherWrapperComparable) {
        return -(integer.compareTo(otherWrapperComparable.getInteger()));
    }

    @Override
    public String toString() {
        return "WrapperComparable("+integer+ ")";
    }
}
```

The code for the `TestWrapperComparable` class could be the following:

```java
import java.util.Arrays;

public class TestWrapperComparable {
    public static void main(String args[]) {
        WrapperComparable[] array = {new WrapperComparable(1942),
          new WrapperComparable(1974), new WrapperComparable(1907)};
```

433

```
            Arrays.sort(array);
            for (WrapperComparable wrapperComparable : array) {
                System.out.println(wrapperComparable);
            }
        }
    }
```

The output of the class is:

```
false
true
true
true
false
```

In fact, `string1` and `string2` are two different strings, so it is correct that the first print instruction, which compares strings with the operator `==` which is based on the reference addresses, prints `false`. The second and third print statements will print `true`, because the `equals()` method compares the contents of the strings and not the reference addresses. The result of the fourth and fifth printing instructions instead, depends on the fact that the string `Claudio` is a string that has been put in the pool of strings (see section 13.5.1) and whose address coincides with the address of `string1` (see the first instruction of the method) but not with that of `string2`.

All statements are correct. In particular in the last one we talk about modifying the external state of an object, but there is no "external state" to modify.

All statements are false! In particular the number 2 and the number 3 are incorrect because a character of a string is stored in 16 bits (not a string), while if the string is compact one of its characters is stored using 8 bits. The number 4 is incorrect because the `-XX: -CompactStrings` option should be used when running the program, not when compiling it.

The correct result would be 13, but the correct answer is the last one. In fact in the last instruction the parentheses are missing to invoke the method `length()`, which causes the following

error in compilation:

```
Exercise13M.java:10: error: cannot find symbol
        System.out.println(string.length);
                                 ^
  symbol:   variable length
  location: variable string of type String
1 error
```

The compiler warns us that it does not find the `length` variable (since the round brackets characterizing the syntax of the methods are missing). Note that all instructions (except the penultimate) do not reassign the result of the method invoked to `string`, which therefore always points to the same immutable object declared initially. Only in the penultimate instruction a new object (which concatenates an exclamation point at the end of the string) is reassigned to the string variable with the `+=` operator. That is why if there were no error in the last statement, the value `13` would have been printed.

Solution 13.n)

The output of the `Exercise13N` class is as follows:

```
intern()==equals()
```

therefore, the correct answer is the number 6. In fact, the call to the method `intern()` tries to retrieve the `String` object on which it is called from the pool of strings, using the comparison that provides the `equals ()` method. If the desired string does not exist in the pool, it is added, and a reference to it is returned. So, the string is added to the pool, and subsequent `if` conditions are checked.

Solution 13.o)

The output of the `Exercise12N` class is as follows:

```
123456
```

Note that the output includes a new line. In fact, after having given as a value `123789` to the variable `string1`, we get `string2` concatenating to `string1` a line separator (which causes a newline) thanks to the static method `lineSeparator()` of the `System` class. Calling the method `toCharArray()` on `string2`, we store in `array1` the array of characters that made up the string `string2`. So this array has size 8 if run on Windows (where `System.lineSeparator()` contains two characters `\r` and `\n`) and has dimension 7 on other platforms like Linux (where

System.lineSeparator() contains only the escape character \n) . Then to the array2 array is assigned three elements always of type character (4, 5 and 6) that are all copied through the System.arraycopy() method in the array1 starting from index 3. Finally, the array array1 is printed and you exit the program using the System.exit() method (superfluous in this case because the program would have terminated the same immediately after).

Solution 13.p)

The solution could be implemented in the following way. We create the following enumeration that simply defines some type of font.

```
public enum Font {
    ARIAL, TIMES_NEW_ROMAN, COURIER, MONOSPACED;
}
```

Then we create the required RTFText class in this way:

```
public class RTFText implements Cloneable {
    String text;
    Font character;
    boolean underlined;

    public RTFText(String text, Font character, boolean underlined) {
        this.text = text;
        this.character = character;
        this.underlined = underlined;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public String toString(){
        return "Text = " + text + ", character = " + character +
            ", underlined = " + underlined;
    }
}
```

Finally, we can write the test class this way:

```
public class Exercise13P {
    public static void main(String args[]) throws CloneNotSupportedException {
        RTFText text = new RTFText("Java", Font.ARIAL, false);
        System.out.println(text);
        System.out.println(text.clone());
    }
}
```

436

That will produce the following output.

```
Text = Java, character = ARIAL,  underlined = false
Text = Java, character = ARIAL,  underlined = false
```

Solution 13.q)

One of the possible solutions is the update of the Authentication class:

```java
package com.claudiodesio.authentication;

public class Authentication {

    public void login() {
        String username = System.getProperty("username");
        System.out.println(username);
        String password = System.getProperty("password");
        System.out.println(password);
        User user = findUser(username);
        if (user != null) {
            if (verifyPassword(user, password)) {
                Print.sayHello(user.getName());
            } else {
                Print.authenticationFailed();
            }
        } else {
            Print.usernameNotFound();
        }
    }

    private User findUser(String username) {
        User[] users = UserProfiles.getInstance().getUsers();
        if (username != null) {
            for (User user : users) {
                if (username.equals(user.getUsername())) {
                    return user;
                }
            }
        }
        return null;
    }

    private boolean verifyPassword(User user, String password) {
        boolean found = false;
        if (password != null) {
            if (password.equals(user.getPassword())) {
                found = true;
            }
        }
        return found;
```

437

```
    }

    public static void main(String args[]) {
        Authentication authentication = new Authentication();
        authentication.login();
    }
}
```

which performed with these parameters:

```
java -Dusername=dansap -Dpassword=music
    com.claudiodesio.autentication.Authentication
```

will result in the following output::

```
dansap
music
Hello Daniele
```

Solution 13.r)

The statements number 3 and 6 are correct. The number 1 is false because other algorithms such as SerialGC can also be used. In the number 2 statement the first part of the sentence is not correct: the ParallelGC interventions are less frequent (but more intense). The "finalization" consists in testing if there are objects no longer reachable by some reference and therefore not usable, therefore also the statement 4 is wrong. The runFinalization() method requires the finalization of the objects, but resides in the Runtime class not in Object, which is why the statement 5 is also incorrect

Solution 13.s)

A possible solution is represented by the following class:

```
import java.lang.reflect.*;

public class Exercise13S {
    public static void main(String args[]) throws Exception {
        Class<ClassWithPrivateMembers> classWithPrivateMembers =
            ClassWithPrivateMembers.class;
        ClassWithPrivateMembers object =
            classWithPrivateMembers.getDeclaredConstructor().newInstance();
        Field privateVariable =
            classWithPrivateMembers.getField("privateVariable");
        privateVariable.setAccessible(true);
        privateVariable.set(object, "Private variable hacked!");
```

```
        System.out.println(privateVariable.get(object))
        Method privateMethod =
            classWithPrivateMembers.getMethod("privateMethod");
        privateMethod.setAccessible(true);
        privateMethod.invoke(object);
    }
}
```

The code is very intuitive, except for the fact that, in setting the variable and invoking the method, it is also necessary to pass the object on which you want to act.

The output of this program stops at the seventh line:

```
Exception in thread "main" java.lang.NoSuchFieldException:
  privateVariable
      at java.base/java.lang.Class.getField(Class.java:1956)
      at Exercise13S.main(Exercise13S.java:7)
```

While if we comment out the lines that trigger the exception and re-run the file (obviously after recompiling it):

```
import java.lang.reflect.*;

public class Exercise13S {
    public static void main(String args[]) throws Exception {
        Class<ClassWithPrivateMembers> classWithPrivateMembers =
            ClassWithPrivateMembers.class;
        ClassWithPrivateMembers object =
            classWithPrivateMembers.getDeclaredConstructor().newInstance();
        /* Field privateVariable =
            classWithPrivateMembers.getField("privateVariable");
        privateVariable.setAccessible(true);
        privateVariable.set(object, "Private variable hacked!");
        System.out.println(privateVariable.get(object));*/
        Method privateMethod =
            classWithPrivateMembers.getMethod("privateMethod");
        privateMethod.setAccessible(true);
        privateMethod.invoke(object);
    }
}
```

we will get the following output:

```
Exception in thread "main" java.lang.NoSuchMethodException:
  ClassWithPrivateMembers.privateMethod()
      at java.base/java.lang.Class.getMethod(Class.java:2065)
      at Exercise13S.main(Exercise13S.java:11)
```

Solution 13.t)

Just use the PI variable (which represents the π of the Math class) and the pow() method (which represents the power function) in the following way:

```
public class Exercise13T {
    public static void main(String args[]) {
        int radius = 7;
        /*INSERT CODE HERE*/
        double area = Math.PI * Math.pow(radius, 2);
        System.out.println("The area of the circumference with radius 7 is "
            + area);
    }
}
```

And this is its output:

```
The area of the circumference with radius 7 is 153.93804002589985
```

Solution 13.u)

The correct answer is 6, in fact the output is the following:

```
Exception in thread "main" java.lang.NullPointerException
        at Eserczio12U.main(Eserczio12U.java:5)
```

This is because the result of calling the floor() method is not reassigned to any variable, and therefore the e variable remains with the initial value (2.718281828459045). Then the check() method returns null, because 2.0 is different from 2.718281828459045. But null cannot be assigned to a boolean primitive type

Solution 13.v)

The output of the Exercise13.v class is as follows:

```
long 128
```

In fact, the value 128, as we stated in Chapter 2, is considered an int default value. Therefore, although the autoboxing is always valid, the method that has as its parameter a primitive type is invoked, for reasons of code backward compatibility, as explained in section 13.6.1.7.

Solution 13.z)

Our solution is to create a Symbol enumeration, which defines the three signs of the game:

```
public enum Symbol {
    ROCK, PAPER, SCISSORS;
}
```

Then we created a class that we called `Rules`, which represents the "business core" of the game. Here is the algorithm that defines the winner. We have defined this algorithm by implementing the `compare()` method of the `Comparator` interface. The choice fell on this method more for educational purposes than for real utility (surely there are better solutions):

```java
import java.util.Comparator;

public class Rules implements Comparator<Symbol> {
    @Override
    public int compare(Symbol symbol1, Symbol symbol2) {
        int result = 0;
        switch (symbol1) {
            case PAPER: {
                if (symbol2 == Symbol.ROCK) {
                    result = 1;
                } else if (symbol2 == Symbol.SCISSORS) {
                    result = -1;
                }
            }
            break;
            case ROCK: {
                if (symbol2 == Symbol.SCISSORS) {
                    result = 1;
                } else if (symbol2 == Symbol.PAPER) {
                    result = -1;
                }
            }
            break;
            case SCISSORS: {
                if (symbol2 == Symbol.PAPER) {
                    result = 1;
                } else if (symbol2 == Symbol.ROCK) {
                    result = -1;
                }
            }
            break;
            default: {
                result = 0;
            }
            break;
        }
        return result;
    }
}
```

The most important class is the RockPaperScissors class, that defines a single public method play(), and three private methods::

- getSymbol() which retrieves an element of the Symbol enumeration based on its position (id).

- getComputerSymbol() which reuses the getSymbol() method by passing a random number between 0 and 2.

- calculateWinner() which returns the string to be printed as program output, based on the comparison defined in the object Rules.

```java
import java.util.Random;

public class RockPaperScissors {
    public void play(int id) {
        Symbol playerSymbol = getSymbol(id);
        Symbol computerSymbol = getComputerSymbol();
        System.out.println(playerSymbol + " VS " + computerSymbol);
        String result = calculateWinner(playerSymbol, computerSymbol);
        System.out.println(result);
    }

    private String calculateWinner(Symbol playerSymbol, Symbol computerSymbol) {
        Rules rules = new Rules();
        int result = rules.compare(playerSymbol, computerSymbol);
        if (result > 0) {
            return playerSymbol + " Wins!\nYou win!";
        } else if (result < 0) {
            return computerSymbol + " Wins!\nYou loose!";
        } else {
            return "Draw!";
        }
    }

    private Symbol getSymbol(int id) {
        Symbol[] symbols = Symbol.values();
        Symbol computerSymbol = symbols[id];
        return computerSymbol;
    }

    private Symbol getComputerSymbol() {
        Random random = new Random();
        return getSymbol(random.nextInt(3));
    }
}
```

Finally the main() class manages any input from the user's command line and calls the

`RockPaperScissors` `play()` method:

```java
import java.util.Random;

public class Exercise13Z {
    public static void main(String args[]) {
        int id = getId(args);
        RockPaperScissors rockPaperScissors = new RockPaperScissors();
        rockPaperScissors.play(id);
    }

    public static int getId(String args[]) {
        int id = 0;
        if (args.length != 0) {
            try {
                id = Integer.parseInt(args[0]);
                if (id < 0 || id > 2) {
                    System.out.println("Enter a number between 0 and 2");
                    System.exit(1);
                }
            } catch (Exception exc) {
                System.out.println("Input not valid: " + args[0]);
                System.exit(1);
            }
        } else {
            id = new Random().nextInt(3);
        }
        return id;
    }
}
```

Here are some examples of output:

```
ROCK VS PAPER
PAPER Wins!
You loose!

SCISSORS VS PAPER
SCISSORS Wins!
You win!

SCISSORS VS SCISSORS
Draw!

SCISSORS VS ROCK
ROCK Wins!
You loose!
```

Chapter 14
Exercises

Utilities API:
`java.util` Package
and Date-Time API

The chapter is essentially dedicated to the `java.util` package and the Date & Time API. The `java.util` package is very large, there are many other classes and interfaces worthy of note. Using the documentation is essential, also to do the following exercises.

Exercise 14.a) Package `java.util`, *True or False:*

1. The `Properties` class extends `Hashtable` but allows you to save key-value pairs in a file, making them persistent.

2. The `Local` class abstracts the concept of "zone".

3. The `ResourceBundle` class represents a properties file that allows you to manage internationalization. The relationship between the file name and the specified locale to locate this file, will allow us to manage the language configuration of our applications.

4. The output of the following code:

```
StringTokenizer st = new StringTokenizer(
    "The object oriented language Java", "t", false);
```

```
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
```

will be:

```
The object
t
orien
t
ed language Java
```

5. The following code is invalid:

```
Pattern p = Pattern.compile("\bb");
Matcher m = p.matcher("blahblahblah...");
boolean b = m.find();
System.out.println(b);
```

6. The `Preferences` class allows you to manage configuration files with an XML file.

7. The `Formatter` class defines the `printf()` method.

8. The regular expression `[a]` is a quantifier (greedy quantifier).

9. The following code:

```
Date d = new Date();
d.now();
```

creates a `Date` object with the current date.

10. A `SimpleNumberFormat` can format and analyze any currency using `Locale`.

Exercise 14.b) Date-Time API, True or False:

1. The "from" type methods allow to recover a certain temporal type, starting from a temporal type with more information.

2. The only way to parse a string to get an `Instant` object is to use the `DateTimeFormatter` class.

3. The `Duration` class calculates the distance between two instants, so it is defined on the timeline.

4. It is not possible to store time information in an object of type `LocalDate`.

5. You can store date information in an object of type `LocalTime`.

6. You can store date information in an object of type `ZonedDate`.

7. The `between()` method of `ChronoUnit` returns a `Duration` object.

8. The temporal adjusters can be passed to "with" methods to perform operations on dates and times.

9. The temporal queries can be passed to "with" methods to retrieve information on dates and times.

10. In general, it is possible to replace the `Date` class with the `Instant` class.

Exercise 14.c)

Create a `Translator` class that exposes a `translate()` method to translate a limited number of words from English to another language and vice versa, using a resource bundle.

Exercise 14.d)

Create a `TranslationTest` class to test the correctness of the translations.

Exercise 14.e)

Create a `StringUtils` class that declares only static methods (and therefore it is useless to instantiate). It must expose a `search()` method, which through regular expressions must search for all words in a text (specified as input) that start with a certain character, and return them within a list.

Exercise 14.f)

Create a `StringUtilsTest` class to test the correctness of the `search()` method.

Exercise 14.g)

Create a `DateUtils` class that declares only static methods (and therefore it is useless to instantiate). This class, given two instants as input, must be able to return the number of:

- seconds
- minutes
- hours
- days

447

■■ weeks

■■ months

that exist between the two instants.

Exercise 14.h)

Modify the `DateUtils` class to provide it with a method that, given an instant as input, must be able to return the number of:

■■ seconds

■■ minutes

■■ hours

■■ days

■■ weeks

■■ months

that exist between the specified time and the current instant.

Exercise 14.i)

In the `DateUtils` class, create a method that returns the correct time in the "`HH:mm ss`" format.

Exercise 14.l)

In the `DateUtils` class, create a method that formats the specified date according to the pattern specified as input.

Exercise 14.m)

In the `DateUtils` class, create a method that analyzes the specified date according to the pattern specified as input, and returns a `LocalDate`.

Exercise 14.n)

Create a `TestDateUtils` class to test the correctness of `DateUtils` methods.

Exercise 14.o)

Create a simple program that simulates a dice roll. Running the program will print a random number from 1 to 6. Use the official documentation to find a way to generate random numbers.

Tip: create just a single class with a `main()` method containing a single statement.

Exercise 14.p)

Let's continue the discussion started in Exercise 5.r, implemented in Exercise 6.z, and formalized in Exercise 7.z, as a use case in the case study called Logos, introduced in section 5.5. In the *authentication* use case, a user entered a username and password to authenticate himself in the system. But who entered the credentials in the system to allow the user to authenticate with username and password? We find the answer in the use case diagram, shown again in Figure 14.p.1.
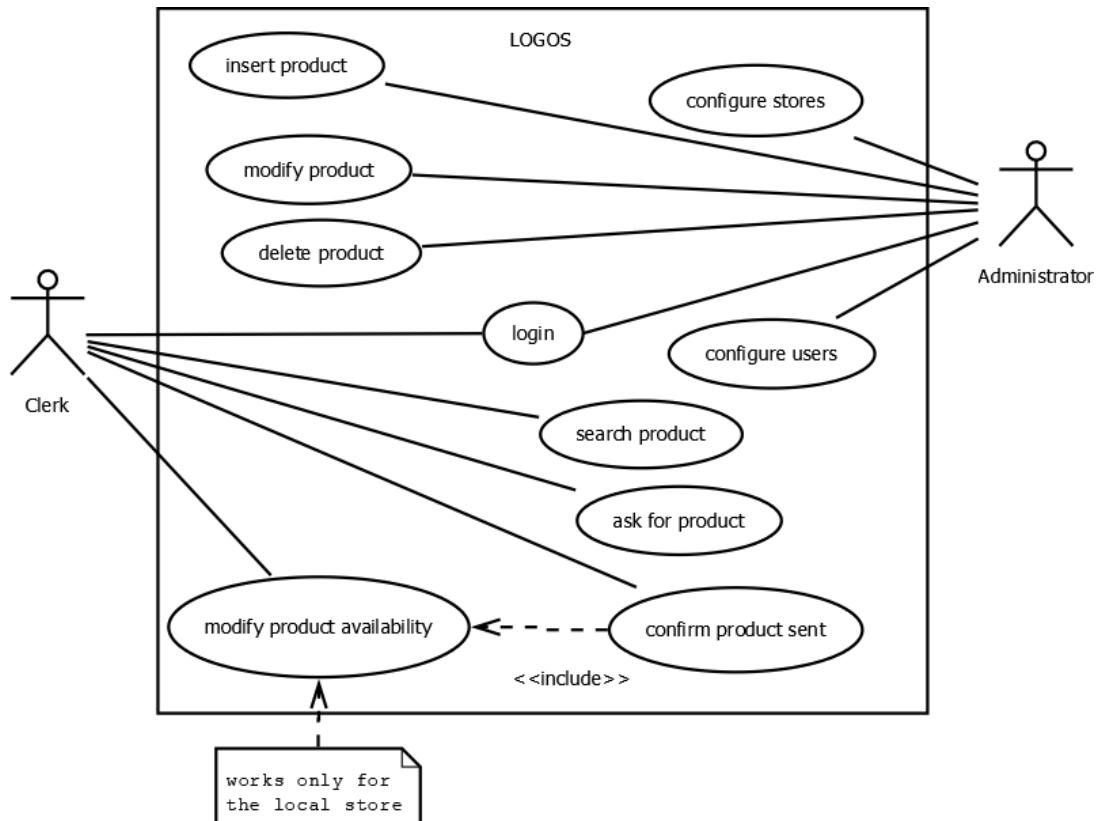
Figure 14.p.1 – The updated Logos use case diagram.

This exercise requires to implement the *configure users* use case. Even this use case must be understood as a small working program, independently of Logos, so that it can possibly be reused in other programs. To be able to store the username and password pairs, use a properties file. Keep in mind that in addition to username and password, the User class must also have other properties to configure, such as the name and role

> **Reuse the code of exercises 5.z and 6.z and of the other related exercises.**

Exercise 14.q)

Keeping in mind the solution of the previous exercise, modify the code of the exercise 6.z and in particular the UserProfiles class, so as to load the User objects starting from the properties file.

Exercise 14.r)

Which of the following statements are correct?

1. A Formatter object allows you to store key-value pairs as a Properties object, without having to specify a file to use for data storage.

2. A System.out object is of type java.io.PrintStream.

3. The java.io.PrintStream class. defines the printf() method which uses the format() method of the Formatter class.

4. The Formatter class defines an overload of the format() method.

5. The Formatter class also allows formatting based on Locale objects.

Exercise 14.s)

Which of the following statements are correct about the Preferences class?

1. A Preferences object allows you to store key-value pairs as a Properties object, without having to specify a file to use for data storage.

2. A Preferences object does not use the setProperty() method.

3. The Preferences class extends HashMap.

4. The `Preferences` class is instantiated without being able to use constructors, but methods that return instances. In fact, it is even an abstract class.

5. The `Preferences` class defines methods to insert specific types of data as a value.

Which of the following statements are correct about regular expressions?

1. The `Matcher` class defines the `find()` method which returns the string that matches the `Matcher` boolean expression.

2. The `Matcher` class is instantiated without being able to use constructors, but methods that return instances.

3. A `Pattern` object defines the `start()` and `end()` methods.

4. Within Java strings, the symbol \ must be repeated, to prevent the compiler from considering it the prefix of an escape character.

5. A "greedy quantifier" is a symbol that represents the multiplicity of occurrences matching a certain pattern.

Which of the following statements are correct regarding the standardization of Date and Time API methods?

1. The "from" methods have an always less "complete" parameter than the type they have to return.

2. The "of" methods return instances of objects on the class on which they are invoked.

3. The "plus" and "minus" methods return copies of the input parameters.

4. "with", "is", "to" and "at" are used only as prefixes, never as complete names.

Which of the following statements are correct regarding the geolocation of the Date and Time API?

1. The `ZoneId` class abstracts the concept of a geographical area that shares the same time zone, and is usually identified by a pair of the "region/city" type.

451

2. The ZoneOffset class abstracts the concept of time zone.

3. The ZoneDateTime class represents the localized version of the LocaleDate class.

4. The atZone() method of the LocalDateTime class returns a ZoneDateTime.

Exercise 14.z)

Which of the following statements are correct regarding the handling of the legacy code of the Date and Time API?

1. The from() method is declared both in the GregorianCalendar class and in the Date class.

2. The toInstant() method is declared in both the Calendar class and the Date class.

3. You can replace the Date class with the Instant class.

4. The GregorianCalendar class can be replaced as appropriate, with the types ZonedDateTime, LocalTime or LocalDate.

5. The TimeZone class can be replaced as appropriate with ZoneId or ZoneOffset types.

Chapter 14
Exercise Solutions

Utilities API:
`java.util` Package
and Date-Time API

1. **True**.

2. **True**.

3. **True**.

4. **False**, all the "t" should not be there.

5. **False**, is valid but will print `false`. In order for you to print `true`, the expression must be changed to "\\bb".

6. **False**, the way persistent data is stored is platform dependent. A `Properties` object, on the other hand, can also use configuration files in XML format.

7. **False**, the `PrintStream` class defines the `printf()` method, the `Formatter` class defines the `format()` method.

8. **False**, is a character class.

9. **False**, the `Date` class does not have a `now()` method. The first line alone would have fulfilled the required task.

10. **False**, the `SimpleNumberFormat` class simply does not exist. But the `SimpleDateFormat` class exists.

Solution 14.b) Date-Time API, True or False:

1. **True**.

2. **False**, also the same `Instant` class defines a `parse()` method.

3. **False**, `Duration` is not connected to the timeline as it represents a time interval between two `Instant` objects.

4. **True**.

5. **False**.

6. **False**, the `ZonedDate` class simply does not exist.

7. **False**.

8. **True**.

9. **False**.

10. **True**.

Solution 14.c)

The code of the `Translator` class could be like the following:

```java
import java.util.Locale;
import java.util.ResourceBundle;

public class Translator {

    private LanguageEnum language;

    private ResourceBundle resources;

    public Translator (LanguageEnum language) {
        this.language = language;
```

```
        String keyLanguage = language.getKey();
        Locale locale = new Locale(keyLanguage);
        resources = ResourceBundle.getBundle("resources.vocabulary", locale);
    }

    public String translate(WordsEnum text) {
        String translation = resources.getString(text.getKey());
        return translation;
    }

    public void setLanguage(LanguageEnum language) {
        this.language = language;
    }

    public LanguageEnum getLanguage() {
        return language;
    }
}
```

Note that to simplify our exercise we have decided to limit the number of words to be translated using the WordsEnum enumeration:

```
public enum WordsEnum {

    BOOK("book"), TIME("time"), HOME("home");

    private String key;

    private WordsEnum(String key) {
        this.key = key;
    }

    public String getKey() {
        return key;
    }
}
```

We have also limited the number of languages supported through the LanguageEnum enumeration:

```
public enum LanguageEnum {

    ITALIAN("it", "Italian"), ENGLISH("en", "English");

    String key;

    String description;

    LanguageEnum(String key, String description) {
```

```
        this.key = key;
        this.description = description;
    }

    public String getKey() {
        return key;
    }

    public String toString() {
        return description;
    }
}
```

We also created in the **resource** folder (inserted in the source folder downloadable from the same address from where you downloaded these files: **http://www.javaforaliens.com/download.html**) the properties files that we needed: **vocabulary_it.properties**:

```
book=libro
home=casa
time=tempo
```

and **vocabolary_en.properties**:

```
book=book
home=home
time=time
```

Solution 14.d)

The code could be the following:

```
public class TranslatorTest {

    public static void main(String args[]) {
        Translator translator = new Translator(LanguageEnum.ENGLISH);
        String translatedWord = translator.translate(WordsEnum.BOOK);
        System.out.println(translatedWord);
    }
}
```

The output will be:

```
book
```

we could also use the Scanner class and make the program interactive.

456

Solution 14.e)

The code could be the following:

```
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class StringUtils {

    public static List<String> search(String text, char firstCharacter) {
        List<String> list = new ArrayList<>();
        Pattern pattern = Pattern.compile("\\b"+firstCharacter+"[a-zA-Z]+\\b");
        Matcher matcher = pattern.matcher(text);
        while (matcher.find()) {
            list.add(matcher.group());
        }
        return list;
    }
}
```

Solution 14.f)

The code of the StringUtilsTest class could be the following:

```
import java.util.List;

public class StringUtilsTest {

    public static void main(String args[]) {
        List<String> list = StringUtils.search(
            "The smile of dawn arrived early May "
            + "she carried a gift from her home "
            + "the night shed a tear to tell her of fear "
            + "and of sorrow and pain she'll never outgrow ", 't');
        for (String string : list) {
            System.out.println(string);
        }
    }
}
```

The output of the previous code is:

```
the
tear
to
tell
```

Solution 14.g)

The code could be the following:

```
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;

public class DateUtils {

    public static long getInterval(LocalDateTime ldt1, LocalDateTime ldt2,
      ChronoUnit chronoUnit) {
        return chronoUnit.between(ldt1, ldt2);
    }
}
```

But there are so many alternatives to this solution.

Solution 14.h)

The code of the requested method is very trivial:

```
    public static long getPastTime(Instant instant1, ChronoUnit chronoUnit) {
        return getInterval(instant1, Instant.now(), chronoUnit);
    }
```

In fact, we have reused the method written in the previous exercise.

Solution 14.i)

The code of the requested method is very trivial:

```
    public static String getExactTime() {
        LocalTime now = LocalTime.now();
        String exactTime = (now.getHour() + ":" + now.getMinute() + " "
            + now.getSecond());
        return exactTime;
    }
```

Solution 14.l)

The code of the requested method is also very simple:

```
    public static String formatDate(LocalDateTime localDateTime, String pattern)
      throws DateTimeException {
        String formattedDate = null;
        try {
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern);
            formattedDate = formatter.format(localDateTime);
```

458

```
        } catch (DateTimeException dateTimeException) {
            dateTimeException.printStackTrace();
            throw dateTimeException;
        }
        return formattedDate;
    }
```

Note that we handled the exception only to print the stack trace, and then we have rethrown it. We have also included the `throws` clause next to the method definition. In this way, those who use this method will know that they could throw the exception in case you specify an incorrect pattern.

Solution 14.m)

The code of the requested method could be the following:

```
public static LocalDate analizzaData(String data, String pattern)
   throws DateTimeParseException {
     LocalDate localDate = null;
     try {
         localDate = LocalDate.parse(data, DateTimeFormatter.ofPattern(pattern));
     } catch (DateTimeParseException dateTimeParseException) {
         dateTimeParseException.printStackTrace();
         throw dateTimeParseException;
     }
     return localDate;
}
```

Note that even in this case we handled the exception only to print the stack trace, and then we have rethrown it. We have also included the `throws` clause next to the method definition. In this way, those who use this method will know that they could throw the exception in case you specify an incorrect pattern. However, the exception this time is of type `DateTimeParseException`.

> **Also in this case, the solution is very simple. Note that it is also possible to use other classes and other methods to achieve the same results as in the last four exercises. The library for managing dates and time is really simple and powerful.**

Solution 14.n)

The required code could be the following:

```
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.temporal.ChronoUnit;

public class DateUtilsTest {

    private static final String DATE_FORMAT = "MM/dd/yy  hh:mm a";

    public static void main(String args[]) {
        final String exactTime = DateUtils.getExactTime();
        System.out.println("Time right now: " + exactTime);
        Instant twoThousand = Instant.parse("2000-01-01T00:00:00.00Z");
        Instant twoThousandAndTen = Instant.parse("2010-01-01T00:00:00.00Z");
        long daysInterval = DateUtils.getInterval(
           twoThousand, twoThousandAndTen, ChronoUnit.DAYS);
        System.out.println("From 1st January 2000 to 1st January 2010 "
           + daysInterval + " days are passed");
        final long minutesPassed =
           DateUtils.getPastTime(twoThousand, ChronoUnit.MINUTES);
        System.out.println(minutesPassed + " minutes " +
           "have passed since January 1st 2010 ");
        LocalDateTime localDateTime = LocalDateTime.now();
        final String formattedDate =
           DateUtils.formatDate(localDateTime, DATE_FORMAT);
        System.out.println("Formatted date: " + formattedDate);
        LocalDate localDate =
           DateUtils.parseDate(formattedDate, DATE_FORMAT);
        System.out.println(localDate);
        System.out.println("Let's throw an exception!");
        localDate = DateUtils.parseDate(formattedDate, "ABC");
    }
}
```

Solution 14.o)

The right class to use is the `java.util.Random` class, with its `nextInt()` method which takes as input an upper limit (excluded), and which has as its lower limit set to 0 (included). Just add the value 1 and you're done. The code could be the following:

```
import java.util.Random;

public class DiceRoll {
  public static void main(String args[]) {
      System.out.println("Dice rolling... " + (1 + new Random().nextInt(6)) + "!");
  }
}
```

460

Let's reuse the User class:

```
package com.claudiodesio.authentication;

public class User {
    private String name;
    private String username;
    private String password;

    public User(String name, String username, String password) {
        this.name = name;
        this.username =username;
        this.password =password;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

The Amministrator class:

```
package com.claudiodesio.authentication;

public class Administrator extends User {
    public Administrator (String name, String username, String password) {
        super(name, username, password);
    }
}
```

The Clerk class:

```
package com.claudiodesio.authentication;

public class Clerk extends User {
    public Clerk(String name, String username, String password) {
        super(name, username, password);
    }
}
```

Then let's modify the UserProfiles class in order to manage user profiles:

```
package com.claudiodesio.authentication;
import java.util.*;
import java.io.*;

public class UserProfiles {

    private static UserProfiles instance;

    private Properties properties;

    private UserProfiles() {
        properties = new Properties();
        try {
            loadProperties();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static UserProfiles getInstance() {
        if (instance == null) {
            instance = new UserProfiles();
        }
        return instance;
    }

    public void loadProperties() throws IOException {
        try (FileInputStream inputStream =
          new FileInputStream("config.properties");) {
            properties.load(inputStream);
        }
    }

    public void addUser(String[] args) throws IOException {
        String role = args[0];
        String name= args[1];
        String username = args[2];
```

```
        String password = args[3];
        // If the role Administrator is not specified
        //a clerk will always be inserted
        User user = (role.equals("Administrator") ?
          new Administrator(name, username, password):
          new Clerk(name, username, password));
        String value = name + "," + role + "," + password;
        properties.setProperty(username, value);
        try (FileOutputStream fos = new FileOutputStream("config.properties")) {
            properties.store(fos, "Configuration File");
        }
        System.out.println("Added property: " + username + "=" + value);
    }
}
```

In particular we have added a method to add user profiles, which allows us to add a line in the properties file whose key is the username, and whose value is a list with other user information separated by commas. Finally, with the following class which takes the values to be entered from the command line, the application is executed:

```
package com.claudiodesio.authentication;

import java.io.*;

public class Exercise14P {

    public static void main(String args[]) throws IOException {
        if (args.length != 4) {
            System.out.println("Specify role, name, username, password");
            System.exit(1);
        }
        UserProfiles.getInstance().addUser(args);
    }
}
```

By running the application, for example without specifying command line arguments, we will get the following output:

```
java com.claudiodesio.authentication.Eexercise14P
Specify role, name, username, password
```

instead specifying the following arguments:

```
java com.claudiodesio.authentication.Eexercise14P Amdministrator
  Claudio desio xxxxxxxx
Added property: desio=Claudio,Amministratore,xxxxxxxx
```

we will get that in the properties file we will find the following content:

```
#Configuration File
#Sun Oct 05 23:47:31 CEST 2019
desio=Claudio,Amministratore,xxxxxxxx
```

Solution 14.q)

One possible solution is to change the UserProfiles class as follows:

```java
package com.claudiodesio.authentication;

import java.util.*;
import java.io.*;

public class UserProfiles {

    private static UserProfiles instance;

    private Properties properties;

    private UserProfiles() {
        properties = new Properties();
        try {
            loadProperties();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static UserProfiles getInstance() {
        if (instance == null) {
            instance = new UserProfiles();
        }
        return instance;
    }

    public void loadProperties() throws IOException {
        try (FileInputStream inputStream =
          new FileInputStream("config.properties");) {
            properties.load(inputStream);
        }
    }

    public void addUser(String[] args) throws IOException {
        String role = args[0];
        String name= args[1];
        String username = args[2];
        String password = args[3];
        // If the role Administrator is not specified
```

464

```
        //a clerk will always be inserted
        User user = (role.equals("Administrator") ?
          new Administrator(name, username, password):
          new Clerk(name, username, password));
        String value = name +","+ role+","+ password;
        properties.setProperty(username, value);
        try (FileOutputStream fos = new FileOutputStream("config.properties")) {
            properties.store(fos,"Configuration File");
        }
        System.out.println("Added property: " + username + "=" + value);
    }

    public User getUser(String username) {
        String value = (String)properties.getProperty(username);
        if (value == null) {
            return null;
        }
        String [] tokens = value.split(",");
        User user = (tokens[1].equals("Administrator") ?
          new Administrator(tokens[0], username, tokens[2]):
          new Clerk(tokens[0], username, tokens[2]));
        return user;
    }
}
```

In particular we have created the getUser() method that we are going to call from the Authentication class, appropriately modified:

```
package com.claudiodesio.authentication;

import java.util.Scanner;

public class Authentication {

    public void login() {
        boolean authorized = false;
        Scanner scanner = new Scanner(System.in);
        do {
            Print.requestUsername();
            String username = scanner.nextLine();
            User user = UserProfiles.getInstance().getUser(username);
            if (user != null) {
                Print.requestPassword();
                String password = scanner.nextLine();
                if (checkPassword(user, password)) {
                    Print.sayHello(user.getName());
                    authorized = true;
                } else {
                    Print.authenticationFailed();
```

```
                }
            } else {
                Print.usernameNotFound();
            }
        } while (!authorized);
    }

    private boolean checkPassword(User user, String password) {
        boolean found = false;
        if (password != null) {
            if (password.equals(user.getPassword())) {
                found = true;
            }
        }
        return found;
    }
}
```

Finally, the `main()` method we moved to the `Exercise14Q`:

```
package com.claudiodesio.authentication;

public class Exercise14Q {
    public static void main(String args[]) {
        Authentication authentication = new Authentication();
        authentication.login();
    }
}
```

Running the application all works as before:

```
Type username.
foo
User not found!
Type username.
desio
Type password.
yyyyyyyy
Authentication failed
Type username.
desio
Type password.
xxxxxxxx
Hello Claudio
```

Solution 14.r)

All statements are true.

Solution 14.s)

All statements are true except number 3. In fact, the `Preferences` class does not extend `Hashtable`, but is an abstract class that extends `Object`. Just read the Java API documentation

Solution 14.t)

The correct statements are the numbers 2, 4 and 5. The number 1 is false because the `find()` method does not return a string but a boolean (the `find()` method is in fact usually used as a loop condition).
The number 3 is false because the `start()` and `end()` methods do not belong to the `Pattern` class but are declared by the `Matcher` class.

Solution 14.u)

All the statements are true except for the number 1. In fact, the "methods have an always more complete parameter with respect to the type they have to return.

Solution 14.v)

All statements are true except number 3. In fact, the `ZoneDateTime` class represents the localized version of the `LocaleDateTime` class, not of `LocalDate`.

Solution 14.z)

All the statements are true!

Chapter 15 Exercises

Thread Management

We saw in Chapter 15 how thread management is a very complex topic. However, the most of the exercises that are presented below, should be quite feasible.

1. A thread is an object instantiated by the `Thread` class or the `Runnable` class.

2. Multithreading is usually a feature of operating systems and not of programming languages.

3. In every application at runtime there is at least one thread running.

4. Apart from the main thread, a thread needs to execute code within an object whose class extends `Runnable` or extends `Thread`.

5. The `run()` method must be called by the programmer to activate a thread.

6. The current thread is not usually identified with the reference `this`.

7. A call to the `start()` method on a thread, means that it is immediately executed.

8. The `sleep()` method is static and allows the thread that reads this instruction to sleep for a specified number of milliseconds.

9. Prioritizing threads is an activity that can produce different results on different platforms.

10. The JVM scheduler does not depend on the platform on which it is run.

Exercise 15.b) Handling Multi-Threading, True or False:

1. A thread abstracts a virtual processor that executes code on certain data.

2. The synchronized keyword can be used both as a modifier of a method and as a modifier of a variable.

3. The monitor of an object can be identified with the synchronized part of the object itself.

4. In order for two threads running the same code and sharing the same data to have no concurrency problems, you need to synchronize the common code.

5. A thread is said to lock an object if it enters its monitor.

6. The wait(), notify() and notifyAll() methods are the main tool for multiple threads communication.

7. The suspend() and resume() methods are currently deprecated.

8. The notityAll() method, invoked on a certain object o1, reawakens from the paused state all the threads that called wait() on the same object. Among these will be executed that was started first with the start() method.

9. Deadlock is a blocking error condition generated by two threads that depend on each other in two synchronized objects.

10. If a thread t1 executes the run() method in the object o1 of class C1, and a thread t2 executes the run() method in the object o2 of the same class C1, the keyword synchronized is useless.

Exercise 15.c)

Simulate with a working code the following situation with what was learned in this chapter.
A group of 10 people is at the astronomical observatory to admire the passage of a comet using the powerful telescope provided by the structure. Only one person at a time can use the telescope, only three minutes per person to complete the observation are allowed. There is no

line, the participants will access the telescope randomly. Each participant will then go through statuses:

■ The "Waiting" status when waiting for his turn. It lasts indefinitely, depending on when the participant's turn begins;

■ The "Observation" status when the participant is observing the sky through the telescope. It lasts exactly 3 minutes (but it is possible to shorten this time to execute the exercise);

■ the "Done" state when the turn is over.

Statuses can be characterized by significant prints.

Exercise 15.d)

Simulate with a working code the following situation with what was learned in this chapter. Suppose we find ourselves at the front office of the municipality that issues the passports. Suppose there are 10 applicants in a line ready to request the document. When your turn comes, you will be given a form to fill in. In order not to block the queue, the next applicant may in the meantime request the service at the same front office. So, all applicants will be given the form to fill in, the line will be very fast, and in parallel several people will be busy filling in the form. Each applicant could take a variable time from 5 to 10 minutes to fill in the form (but it is possible to shorten this time to execute the exercise), the first one that will finish (regardless of its position in the initial line) can request the printing of its passport. This will be printed in 3 minutes (or 3 seconds if you want!) anyway.

Exercise 15.e)

Consider the following classes:

```java
import java.util.ArrayList;

class RunnableArrayList extends ArrayList implements Runnable {

    public void run(String string) {
        System.out.println("Within the run() method: " + string);
    }
}
public class Exercise15E {
    public static void main(String args[]) {
        RunnableArrayList g = new RunnableArrayList();
        Thread t = new Thread(g);
        t.start();
    }
}
```

If we execute the `Exercise15E` class, what will the output be?

1. `Within the run() method: null`

2. `Within the run() method:`

3. The code is interrupted by an exception in the `Exercise15E` class.

4. The code is interrupted by an exception in the `RunnableArrayList` class.

5. No output. The code does not compile due to an error in the `Exercise15E` class.

6. No output. The code does not compile due to an error in the `RunnableArrayList` class.

Exercise 15.f)

Which of the following statements are correct:

1. A class, to create objects that have a monitor, must extend `Thread`.

2. A class, to create objects that have a monitor, must implement `Runnable`.

3. When "a thread enters the object monitor" then it has the lock of that object. This means that no other thread can enter the monitor of that object.

4. The monitor class is defined by the synchronized methods of a class.

Exercise 15.g)

Which of the following statements are correct?

1. The `Thread` class extends `Runnable`.

2. `Runnable` is a functional interface.

3. The `Thread` class defines an empty `run()` method.

4. The `Thread` class defines the `wait()` and `notify()` methods.

Exercise 15.h)

Create a `Countdown` class which, once activated, starts a countdown from 10 to 0 before ending.

Exercise 15.i)

Which of the following statements are correct:

1. To instantiate a thread just use its constructors.

2. Specifying thread priority is not an effective way to decide the order of execution of multiple threads.

3. To execute the `run()` method defined in a class that extends `Thread`, just call its `run()` method.

4. To execute a thread you need to invoke the `start()` method of the `Object` class.

5. In each program there are at least three threads running.

Exercise 15.l)

Create a program that simulates the following situation. Through interactive commands ("run", "walk", "stop" and "end") read by the `Scanner` class, the user will play the part of the coach of a virtual runner, giving commands on the fly while the runner trains.

Then create a `Runner` class that extends `Thread`. It must declare the methods: `startRunning()` which will allow it to simulate the runner to go fast, `walk()` which will allow it to simulate a walk, `takeABreath()` which will allow it to simulate the action of stopping, and `stopTraining()` (that will end the training (and the program).

> **Tip: we can use boolean variables to manage an infinite cycle that manages the training cycle.**

Exercise 15.m)

Is it better to implement `Runnable` or extend the `Thread` class? Choose all the correct statements.

1. By implementing `Runnable` which is an interface, we can also extend other classes.

2. By extending the `Thread` class we can also implement other interfaces.

3. From the point of view of data abstraction, an object of type `Thread` should not have private variables that represent the data to be managed.

4. A `Runnable` object that implements the `run()` method can be the input to the constructor of a `Thread` object. If the `start()` method is invoked on this last one, the thread that executes the `run()` method is activated.

Which of the following statements are correct?

1. With time slicing (or round-robin) scheduling a thread may be running only for a certain period of time.

2. Time slicing (or round-robin) scheduling is the default behavior of most Linux systems.

3. Preemptive scheduling is the default behavior of most Linux systems.

4. With preemptive scheduling, priority is a more deterministic element than round-robin. In fact launched two threads at the same time, the highest priority thread will enter the execution state, and will exit only when it has finished its work, or if it is called on it a method like `wait()` or `suspend()`, or when must wait for external resources (as in the case of waiting for input-output resources).

Which of the following statements are correct?

1. The `volatile` modifier can only be used on methods and variables.

2. Declaring `volatile` an instance variable implies that all other variables of the same instance that are used by the same thread will also be considered volatile.

3. For a `volatile` variable, all read and write accesses are atomic.

4. For a non-`volatile` integer variable, all read and write accesses are atomic.

Which of the following statements are correct?

1. The `synchronized` keyword allows to declare atomic methods.

2. The monitor of an object consists of the part of the synchronized object. So, if at a certain moment, a thread is executing code within one of the synchronized methods of that

object, any other thread that wants to access the code of any of the synchronized methods of the same object, will have to wait for the first thread to terminate execute synchronized code.

3. You can use synchronized blocks to make only certain lines of code synchronized. This represents a more flexible solution than the static synchronization of a method.

4. You can use synchronized as a modifier only with methods.

5. The run() method can be declared synchronized.

Exercise 15.q)

Which of the following statements are correct?

1. The wait(), notify() and notifyAll() methods are defined in the Thread class.

2. If a thread encounters the wait() method, it release the object monitor of which it was executing code.

3. The notifyAll() method restarts all the threads that had executed the wait() method.

4. The suspend() and resume() methods are defined in the Thread class.

Exercise 15.r)

Make the following class immutable:

```
import java.util.Date;
public final class Exercise15R {
    private final Integer integer;
    private final Date date;
    public Exercise15R(Integer integer, Date date) {
        this.integer = integer;
        this.date = (Date)date.clone();
    }
    public final Date getStringBuilder() {
        return (Date)date.clone();
    }
    public final Integer getInteger() {
        return integer;
    }
}
```

Exercise 15.s)

Which of the following statements are correct about `java.util.concurrent.atomic` and `java.util.concurrent.locks` packages?

1. `ReentrantLock` is a class that, if instantiated, can replace the use of a synchronized block. However, a `try-catch-finally` block must be used.

2. The concept of *fairness* of `ReentrantLock` type objects allows the JVM to create a thread execution priority research, based on the creation time of the `ReentrantLock`.

3. A `Lock` object can specify a timeout to exit a synchronized block.

4. The `AtomicInteger` interface defines atomic methods that perform more than one operation.

Exercise 15.t)

Consider the following snippet:

```
Callable<String> callable = new Callable<>() {public void call(){}};
Future<String> future = Executors.newFixedThreadPool(3).start(callable);
String result = future.get();
```

What message the compiler will return?

Exercise 15.u)

Abstract an alarm clock with the `Timer` and `TimerTask` classes of the `java.util` package. In particular, it must be possible to pass from the command line a number that will represent the number of seconds that must pass in order to make "sound" the alarm.

Exercise 15.v)

Which of the following statements are correct?

1. The statement:

   ```
   new Semaphore().acquire();
   ```

   allows a semaphore to acquire a lock on an object.

2. The `tryAcquire()` method allows you to specify a timeout.

3. `Semaphore` is an interface.

4. The *permits* represent a `Semaphore` instance variables.

476

Exercise 15.z)

Which of the following statements are correct?

1. Instantiated a `cyclicBarrier` object of type `CyclicBarrier`, the statement:

2. `cyclicBarrier.await();`

3. is equivalent to calling the `wait()` method on the object that reads this statement.

4. The `CyclicBarrier` constructor allows you to specify the number of threads that must "accumulate in a certain code point" before being released.

5. The `signalAll()` method of `CyclicBarrier` is equivalent to the object `notifyAll()` method.

6. The use of `CyclicBarrier` can always replace the use of `wait()`, `notify()` and `notifyAll()`.

Chapter 15
Exercise Solutions

Thread Management

1. **False**, Runnable is an interface.

2. **True**.

3. **True**, the so-called *main thread*.

4. **True**.

5. **False**, the programmer can invoke the start() method and the scheduler will invoke the run() method.

6. **True**.

7. **False**.

8. **True**.

9. **True**.

10. **False**.

Solution 15.b) Handling Multi-Threading, True or False:

1. **True**.

2. **False**.

3. **True**.

4. **False**.

5. **True**.

6. **True**.

7. **True**.

8. **False**, the first thread that will start will be the one with the highest priority.

9. **True**.

10. **True**.

Solution 15.c)

First, we abstract the concept of status discussed in question 15.c, through an enumeration. We also create the message to be printed for each status:

```java
package com.claudiodesio.observatory.metadata;

public enum Status {
    WAITING("\"I'm waiting...\""),
    OBSERVATION("\"It's my turn... how wonderful!\""),
    DONE("\"I'm done.\"");

    private String message;

    private Status(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

Then we create a simple class `Participant`, which has among its instance variables a `Status` object. All participants have a name and a telescope to which they refer for observation. Our class extends `Thread`, and its `run()` method defines the observation action:

```
package com.claudiodesio.observatory.data;

import com.claudiodesio.observatory.metadata.Status;

public class Participant extends Thread {

    private Status status;

    private final Telescope telescope;

    public Telescope getTelescope() {
        return telescope;
    }

    public Participant(String name, Telescope telescope) {
        setName(name);
        this.telescope = telescope;
        this.setStatus(Status.WAITING);
        status();
    }

    public Status getStatus() {
        return status;
    }

    public void setStatus(Status status) {
        this.status = status;
    }

    @Override
    public void run() {
        telescope.allowObservation(this);
    }

    public void status(){
        System.out.println(getName() + " says: " + status.getMessage());
    }
}
```

Note that when a `Participant` is created, the status is set to `WAITING`.
Then we create the most important class, that is the one that abstracts the `Telescope` concept:

```
package com.claudiodesio.observatory.data;

import com.claudiodesio.observatory.metadata.Status;

public class Telescope {

    public synchronized void allowObservation(Participant participant) {
```

```
        participant.setStatus(Status.OBSERVATION);
        participant.status();
        try {
            Thread.sleep(3000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        participant.setStatus(Status.DONE);
        participant.status();
    }
}
```

In his synchronized business method, he merely passes from one state to another after a 3 second pause.

Finally, a test class follows:

```
package com.claudiodesio.observatory.test;

import com.claudiodesio.observatory.data.Participant;
import com.claudiodesio.observatory.data.Telescope;

public class Observation {

    public static void main(String args[]) {
        Telescope telescope = new Telescope();
        Participant[] participants = getParticipants(telescope);
        for (Participant participant : participants) {
            participant.start();
        }
    }

    private static Participant[] getParticipants(Telescope telescope) {
        Participant[] participants = {
            new Participant("Ciro", telescope),
            new Participant("Gianluca", telescope),
            new Participant("Pierluigi", telescope),
            new Participant("Gigi", telescope),
            new Participant("Nicola", telescope),
            new Participant("Pino", telescope),
            new Participant("Maurizio", telescope),
            new Participant("Raffaele", telescope),
            new Participant("Fabio", telescope),
            new Participant("Vincenzo", telescope)};
        return participants;
    }
}
```

The output is as follows (which changes with each launch):

482

```
Ciro says: "I'm waiting..."
Gianluca says: "I'm waiting..."
Pierluigi says: "I'm waiting..."
Gigi says: "I'm waiting..."
Nicola says: "I'm waiting..."
Pino says: "I'm waiting..."
Maurizio says: "I'm waiting..."
Raffaele says: "I'm waiting..."
Fabio says: "I'm waiting..."
Vincenzo says: "I'm waiting..."
Ciro says: "It's my turn... how wonderful!"
Ciro says: "I'm done."
Vincenzo says: "It's my turn... how wonderful!"
Vincenzo says: "I'm done."
Fabio says: "It's my turn... how wonderful!"
Fabio says: "I'm done."
Raffaele says: "It's my turn... how wonderful!"
Raffaele says: "I'm done."
Pierluigi says: "It's my turn... how wonderful!"
Pierluigi says: "I'm done."
Gigi says: "It's my turn... how wonderful!"
Gigi says: "I'm done."
Gianluca says: "It's my turn... how wonderful!"
Gianluca says: "I'm done."
Maurizio says: "It's my turn... how wonderful!"
Maurizio says: "I'm done."
Pino says: "It's my turn... how wonderful!"
Pino says: "I'm done."
Nicola says: "It's my turn... how wonderful!"
Nicola says: "I'm done.
```

Solution 15.d)

We create a `TimeUtils` class that defines a utility method to generate a random number between 5 and 10:

```java
package com.claudiodesio.frontoffice.data;

import java.util.*;

public class TimeUtils {

    private static final Random RANDOM = new Random();

    public static int getRandomNumber() {
        return (RANDOM.nextInt(6) + 5);
    }
}
```

The class `Applicant` extends `Thread`:

483

```
package com.claudiodesio.frontoffice.data;

public class Applicant extends Thread {

    public Applicant(String name) {
        setName(name);
    }

    @Override
    public void run() {
        FrontOffice.getInstance().handleRequest(this);
    }

    @Override
    public String toString() {
        return getName();
    }
}
```

The Printer class will be used to print documents:

```
package com.claudiodesio.frontoffice.data;

public class Printer {

    private static Printer instance;

    private Printer() {
    }

    public static Printer getInstance() {
        if (instance == null) {
            instance = new Printer();
        }
        return instance;
    }

    public synchronized void print(Applicant applicant) {
        System.out.println("Employee says: printing passport for " + applicant
          + " in progress...");
        try {
            wait(3000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("Employee says: Print completed! " + applicant
          + " thanks and goodbye!");
    }
}
```

The FrontOffice class is the key class:

```java
package com.claudiodesio.frontoffice.data;

public class FrontOffice {

    private final Printer printer;
    private static FrontOffice instance;

    public synchronized static FrontOffice getInstance() {
        if (instance == null) {
            instance = new FrontOffice();
        }
        return instance;
    }

    private FrontOffice() {
        printer = Printer.getInstance();
    }

    public synchronized void handleRequest(Applicant applicant) {
        System.out.println("Hello " + applicant);
        System.out.println("Employee says: \"Please, fill in the form "
                + applicant + "\"");
        fillInModule(applicant);
        printer.print(applicant);
        System.out.println(applicant + " says: \"Thanks to you!\"");
    }

    private synchronized void fillInModule(Applicant applicant) {
        System.out.println(applicant + " says: \"OK, I will fill it but...\"");
        final int waiting = TimeUtils.getRandomNumber();
        try {
            System.out.println("...I need " + waiting + " minutes...");
            wait(waiting * 1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println(applicant +" says: \"...the module is completed!\"");
    }
}
```

The test class follows:

```java
package com.claudiodesio.frontoffice.test;

import com.claudiodesio.frontoffice.data.Applicant;

public class FrontOfficeTest {
```

```
    public static void main(String args[]) {
        final Applicant[] richiedenti = getApplicants();
        for (Applicant applicant : richiedenti) {
            applicant.start();
        }
    }

    private static Applicant[] getApplicants() {
        Applicant[] applicants = {
            new Applicant("Ciro"),
            new Applicant("Mario"),
            new Applicant("Massimo"),
            new Applicant("Chicco"),
            new Applicant("Enrico"),
            new Applicant("Lorenzo"),
            new Applicant("Emanuele"),
            new Applicant("Cosimo"),
            new Applicant("Alessandro"),
            new Applicant("Salvatore")};
        return applicants;
    }
}
```

Finally, the output follows (which always changes):

```
Hello Alessandro
Employee says: "Please, fill in the form Alessandro"
Alessandro says: "OK, I will fill it but..."
...I need 5 minutes...
Hello Chicco
Employee says: "Please, fill in the form Chicco"
Chicco says: "OK, I will fill it but..."
...I need 6 minutes...
Hello Massimo
Employee says: "Please, fill in the form Massimo"
Massimo says: "OK, I will fill it but..."
...I need 10 minutes...
Hello Mario
Employee says: "Please, fill in the form Mario"
Mario says: "OK, I will fill it but..."
...I need 8 minutes...
Hello Enrico
Employee says: "Please, fill in the form Enrico"
Enrico says: "OK, I will fill it but..."
...I need 5 minutes...
Hello Cosimo
Employee says: "Please, fill in the form Cosimo"
Cosimo says: "OK, I will fill it but..."
...I need 5 minutes...
Hello Emanuele
Employee says: "Please, fill in the form Emanuele"
Emanuele says: "OK, I will fill it but..."
```

```
...I need 5 minutes...
Hello Salvatore
Employee says: "Please, fill in the form Salvatore"
Salvatore says: "OK, I will fill it but..."
...I need 9 minutes...
Hello Lorenzo
Employee says: "Please, fill in the form Lorenzo"
Lorenzo says: "OK, I will fill it but..."
...I need 6 minutes...
Hello Ciro
Employee says: "Please, fill in the form Ciro"
Ciro says: "OK, I will fill it but..."
...I need 9 minutes...
Alessandro says: "...the module is completed!"
Employee says: printing passport for Alessandro in progress...
Employee says: Print completed! Alessandro thanks and goodbye!
Alessandro says: "Thanks to you!"
Mario says: "...the module is completed!"
Employee says: printing passport for Mario in progress...
Employee says: Print completed! Mario thanks and goodbye!
Mario says: "Thanks to you!"
Lorenzo says: "...the module is completed!"
Employee says: printing passport for Lorenzo in progress...
Employee says: Print completed! Lorenzo thanks and goodbye!
Lorenzo says: "Thanks to you!"
Chicco says: "...the module is completed!"
Employee says: printing passport for Chicco in progress...
Employee says: Print completed! Chicco thanks and goodbye!
Chicco says: "Thanks to you!"
Emanuele says: "...the module is completed!"
Employee says: printing passport for Emanuele in progress...
Employee says: Print completed! Emanuele thanks and goodbye!
Emanuele says: "Thanks to you!"
Enrico says: "...the module is completed!"
Employee says: printing passport for Enrico in progress...
Employee says: Print completed! Enrico thanks and goodbye!
Enrico says: "Thanks to you!"
Cosimo says: "...the module is completed!"
Employee says: printing passport for Cosimo in progress...
Employee says: Print completed! Cosimo thanks and goodbye!
Cosimo says: "Thanks to you!"
Massimo says: "...the module is completed!"
Employee says: printing passport for Massimo in progress...
Employee says: Print completed! Massimo thanks and goodbye!
Massimo says: "Thanks to you!"
Ciro says: "...the module is completed!"
Employee says: printing passport for Ciro in progress...
Employee says: Print completed! Ciro thanks and goodbye!
Ciro says: "Thanks to you!"
Salvatore says: "...the module is completed!"
Employee says: printing passport for Salvatore in progress...
Employee says: Print completed! Salvatore thanks and goodbye!
Salvatore says: "Thanks to you!"
```

487

Solution 15.e)

The correct answer is the number 6, since the RunnableArrayList class does not correctly implement the Runnable interface, whose run() method does not take any type of input parameters. In fact, the compilation output is as follows (there are also two warnings relating to the use and declaration of raw type):

```
Exercise15E.java:3: warning: [rawtypes] found raw type: ArrayList
class RunnableArrayList extends ArrayList implements Runnable {
                                ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
D:\claudiodesio.com\Manuale\Java for Aliens 13\Code\chapter_15\exercises\15.e\
 Exercise15E.java:3: error: RunnableArrayList is not abstract and does not override
 abstract method run() in Runnable
class RunnableArrayList extends ArrayList implements Runnable {
^
D:\claudiodesio.com\Manuale\Java for Aliens 13\Code\chapter_15\exercises\15.e\
 Exercise15E.java:3: warning: [serial] serializable class RunnableArrayList has no
 definition of serialVersionUID
class RunnableArrayList extends ArrayList implements Runnable {
^
1 error
2 warnings
```

Solution 15.f)

Only answer 3 is correct, all others not.
In particular, regarding the answer 4, the Monitor class does not exist.

Solution 15.g)

The correct answers are the number 2 and the number 3.
The number 1 is false because the Thread class implements Runnable. In fact, the latter is an interface and as such it must be implemented and not extended by another class.
The number 4 is wrong since the wait() and notify() methods are declared in the class Object.

Solution 15.h)

The Countdown class could be the following:

```
public class Countdown {
    public void run(int seconds) throws InterruptedException {
        for (int i = seconds; i > 0; i--){
            System.out.println(i);
            Thread.sleep(1000);
        }
        System.out.println("Time out!");
    }
}
```

Then we can create a class that texts it:

```
public class Exercise15H {
    public static void main(String args[]) throws Exception {
        Countdown countdown = new Countdown();
        int seconds = 10;
        if(args.length > 0) {
            try {
                seconds = Integer.parseInt(args[0]);
            }
            catch (Exception exc) {
                System.out.println("The input must be a positive " +
                    "integer number, now using the default value 10...");
            }
        }
        countdown.run(seconds);
    }
}
```

By executing the latter without specifying parameters we will get the following output:

```
10
9
8
7
6
5
4
3
2
1
Time out!
```

If we pass parameter 3 then the output will be limited to:

```
3
2
1
Time out!
```

If instead we specify a letter (suppose F) we will have the following output:

```
The input must be a positive integer number, let's use the 10 default value...
10
9
8
7
6
5
4
3
2
1
Time out!
```

Finally, by specifying a negative number or 0, we will get the following output directly:

```
Time out!
```

Solution 15.i)

The correct answers are the numbers 1, 2 and 3. In particular, the 3 does not assert that the run() method is executed in a separate thread, otherwise it would have been wrong. The run() method, however, remains a method, and therefore is invocable like any other method. The statement number 4 is incorrect because the start() method is declared by the Thread class, and not by the Object class. The statement number 5 is incorrect because the right sentence would be: "in every program there is at least one thread in execution".

Solution 15.l)

The specifications of the 15.l exercise are deliberately not too detailed, to leave more space for the reader's imagination. So, in this case a possible solution may differ very much from the solution we present below.

For example we could implement the Runner class in the following way:

```java
public class Runner extends Thread {

    private boolean goAhead;

    private boolean inAction;

    private int gap;

    public Runner() {
        inAction = true;
        gap = 1000;
    }
```

```java
    public void run() {
        while (inAction) {
            try {
                Thread.sleep(gap);
                if (goAhead) {
                    System.out.println("|");
                    Thread.sleep(gap);
                    System.out.println("  |");
                }
            } catch (InterruptedException exc) {
                assert false;
            }
        }
    }

    public void startTraining() {
        start();
    }

    public void startRunning() {
        System.out.println("Ok, let's go...");
        gap = 400;
        goAhead = true;
    }

    public void takeABreath() {
        System.out.println("Ok, I'm standing here.");
        System.out.println("| |");
        goAhead = false;
    }

    public void walk() {
        System.out.println("Ok, I'm, walking...");
        gap = 1000;
        goAhead = true;
    }

    public void stopTraining(){
        System.out.println("Good! I couldn't take it anymore...");
        inAction = false;
    }

    public void cannotUnderstand(String command){
        System.out.println("Sorry! I cannot understand the command "
            + command +  "\nPlease use only run, walk, stop or end");
        inAction = true;
    }
}
```

The core of the business logic lies precisely in the run() method, which with a loop based on the inAction variable, prints with pipe symbols | a trace that looks like steps. Note that when runner have to run, the gap variable, which represents the wait from one step to another, is 400 milliseconds, while when you have to walk the gap is lengthened to a second. Depending on whether the startRunning() method is called or the walk() method then, these steps will be faster or less fast.

But let's see now how we could implement the main class Exercise15L:

```java
import java.util.Scanner;

public class Exercise15L {
    public static void main(String args[]) {
        Runner runner = new Runner();
        runner.startTraining();
        Scanner scanner = new Scanner(System.in);
        boolean loop = true;
        System.out.println(
            "Hi coach, the runner is at your disposal!");
        System.out.println("Write the commands then press the Enter key");
        System.out.println("(run, walk, stop, end)");
        while (loop) {

            String command = scanner.nextLine();
            switch (command) {
                case "run":
                    runner.startRunning();
                break;
                case "walk":
                    runner.walk();
                break;
                case "stop":
                    runner.takeABreath();
                break;
                case "end":
                    runner.stopTraining();
                    loop = false;
                break;
                default:
                    runner.cannotUnderstand(command);
                break;
            }
        }
        try {
            Thread.sleep(2000);
        }
        catch (Exception exc) {
            assert false;
```

```
          }
          System.out.println("End of the training");
      }
  }
```

The code does not differ much from anything we have already seen in these exercises. Through a Scanner object the commands are read, which are translated into calls to the Runner methods.

If we execute this last class we could get the following output, which does not give a good idea if we do not see it in action "live" (better to execute the classes already ready from the **Code\chapter_15\exercises\15.l** of the file of exercises that you have probably already downloaded together with these exercises at **http://www.javaforaliens.com**):

```
Hi coach, the runner is at your disposal!
Write the commands then press the Enter key
(run, walk, stop, end)
walk
Ok, I'm, walking...
|
   |
run|

Ok, let's go...
   |
|
   |
|
   |
|
   |
|
s  |
top|

Ok, I'm standing here.
| |
   |
run
Ok, let's go...
|
   |
w|
al  |
k|
   |

Ok, I'm, walking...
|
   |
hhh|
hhhhhh   |
```

```
h
Sorry! I cannot understand the command hhhhhhhhhh
Please use only run, walk, stop or end
|
  |
dfoghdfgdas|

Sorry! I cannot understand the command dfoghdfgdas
Please use only run, walk, stop or end
  |
stop|

Ok, I'm standing here.
| |
  |
run
Ok, let's go...
|
  |
|
  |
|
  |
|
s  |
top|
  |

Ok, I'm standing here.
| |
end
Good! I couldn't take it anymore...
End of the training
```

Note that, the commands entered, often extend over several lines, because in the meantime the program is printing "the steps".

Solution 15.m)

All the statements are correct except for the number 2. In fact, even by implementing the Runnable interface you have the possibility to implement other interfaces.

Solution 15.n)

All the statements are correct except the number 3. In fact, the preemptive scheduling is the default behavior of most Unix systems.

Solution 15.o)

All statements are correct except number 1. In fact, the volatile modifier can only be used on instance variables.

494

Solution 15.p)

All statements are correct..

Solution 15.q)

All statements are correct except number 1. In fact, the `wait()`, `notify()` and `notifyAll()` methods are defined int the `Object` class.

Solution 15.r)

A possible solution is the following:

```java
import java.util.Date;
public final class Exercise15R {

    private final Integer integer;

    private final Date date;

    public Exercise15R(Integer integer, Date date){
        this.integer = integer;
        this.date = (Date)date.clone();
    }

    public final Date getStringBuilder() {
        return (Date)date.clone();
    }

    public final Integer getInteger() {
        return integer;
    }
}
```

Solution 15.s)

All statements are correct except number 4. In fact, `AtomicInteger` is a class not an interface.

Solution 15.t)

Compiling the snippet will cause three errors, one for each line:

```
Exercise15T.java:6: error: <anonymous Exercise15T$1> is not abstract and does not
 override abstract method call() in Callable
        Callable<String> callable = new Callable<>() {public void call(){}};
                                                       ^
Exercise15T.java:6: error: call() in <anonymous Exercise15T$1> cannot
```

```
 implement call() in Callable
        Callable<String> callable = new Callable<>() {public void call(){}};
                                                                          ^
  return type void is not compatible with String where V is a type-variable:
    V extends Object declared in interface Callable
Exercise15T.java:7: error: cannot find symbol
        Future<String> future = Executors.newFixedThreadPool(3).start(callable);
                                                                      ^
  symbol:   method start(Callable<String>)
  location: interface ExecutorService
3 errors
```

In fact, in the first line we tried to create an anonymous class that extends `Callable` parameterized with a string. Then the `call()` method is called which returns `void` instead of `String`, which however does not exist. In the second line the error consists in calling the `start()` method (as for `Runnable`), but the method to call is `submit()`. Finally, in the third line, the method `get()` of `Future` is invoked which requires the handling of the `InterruptedException`. Thus, the correct code would be similar to the following:

```java
Callable<String> callable = new Callable<>() {
    public String call(){return "";}
};
ExecutorService service = Executors.newFixedThreadPool(3);
Future<String> future = service.submit(callable);
String result = null;
try {
    result = future.get();
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

Solution 15.u)

A possible (and simple) solution could be the following:

```java
import java.util.*;
import java.time.*;
import java.text.*;

public class Exercise15U extends TimerTask {

    private Timer timer;

    public Exercise15U () {
        timer = new Timer();
    }
```

```
    @Override
    public void run() {
        DateFormat timeFormatter =
            DateFormat.getTimeInstance(DateFormat.DEFAULT, Locale.getDefault());
        System.out.println("Wake Up! It's "+ LocalTime.now());
        timer.cancel();
    }

    public static void main(String args[]) throws Exception {
        int seconds = Integer.parseInt(args[0])*1000;
        Exercise15U timerTask = new Exercise15U();
        timerTask.timer.schedule(timerTask, seconds);
    }
}
```

Solution 15.v)

The correct answers are the number 2 and the number 4.

The number 1 defines a incorrect statement, because at least the number of permits must be specified for the Semaphore constructor.

The number 3 is false because Semaphore is a class (easy to guess since it is instantiated with a constructor).

Solution 15.z)

The correct answers are the numbers 1 and 2.

The number 3 is false because signalAll() is not a CyclicBarrier method but belongs to Condition, which we mentioned in the last lines of section 15.6.3.1.

The number 4 is false, because CyclicBarrier only provides the possibility for a group of threads to synchronize and expect each other at a certain point in the code.

Chapter 16
Exercises

Annotation Types

With these exercises we will try to understand what it means to take advantage of the annotation types. We will create an annotation from scratch, we will exploit it with an ad hoc application. We will create a code checker, which based on annotations will decide if our classes meet certain requirements. Then many other exercises that support certification preparation will follow, with multiple choice quizzes.

> Until now, to simplify our work we have made little use of packages, which are usually always used. With annotations we will always use the packages for our exercises. This is necessary because annotations are not detectable by reflection if they are not in a package. So, we recommend using an IDE like Eclipse or Netbeans.

Exercise 16.a) Annotations, Declaration and Usage, True or False:

1. An annotation is a modifier.

2. An annotation is an interface.

3. The elements of an annotation seem abstract methods but imply an implicit implementation.

4. The following is a valid annotation statement:

```
public @interface MyAnnotation {
    void method();
}
```

5. The following is a valid annotation statement:

```
public @interface MyAnnotation {
    int method(int value) default 5;
}
```

6. The following is a valid annotation statement:

```
public @interface MyAnnotation {
    int method() default -99;
    enum MyEnum{TRUE, FALSE};
    MyEnum myEnum();
}
```

7. Suppose the `MyAnnotation` annotation defined in point 6 is correct. With the following code it is used correctly:

```
public @MyAnnotation (
    MyAnnotation.MyEnum.TRUE
)
MyAnnotation.MyEnum m() {
    return MyAnnotation.MyEnum.TRUE;
}
```

8. Suppose the `MyAnnotation` annotation defined in point 6 is correct. With the following code it is used correctly:

```
public @MyAnnotation (
    myEnum=MyAnnotation.MyEnum.TRUE
)
MyAnnotation.MyEnum m() {
    return @MyAnnotation.myEnum;
}
```

9. Consider the following annotation.

```
public @interface MyAnnotation {
    int myValue();
}
```

With the following code it is used correctly:

```
public @MyAnnotation (
    5
)
```

500

```
void  m()
    //...
}
```

10. Consider the following annotation:

```
public @interface MyAnnotation {}
```

With the following code it is used correctly:

```
public @MyAnnotation void  m() {
    //...
}
```

Exercise 16.b) Annotations API, True or False:

1. The following annotation is also a meta-annotation:

```
public @interface MyAnnotation ()
```

2. The following annotation is also a meta-annotation:

```
@Target(ElementType.SOURCE)
public @interface MyAnnotation ()
```

3. The following annotation is also a meta-annotation:

```
@Target (ElementType.@INTERFACE)
public @interface MyAnnotation ()
```

4. The following annotation, if applied to a method, will be documented in the relevant Javadoc documentation:

```
@Documented
@Target (ElementType.ANNOTATION_TYPE)
public @interface MyAnnotation ()
```

5. The following annotation will be inherited if and only if applied to a class:

```
@Inherited
@Target (ElementType.METHOD)
public @interface MyAnnotation ()
```

6. For the following annotation it is also possible to create an annotation processor that recognizes the type of annotation at runtime, to implement a particular behavior:

```
@Documented
@Target (ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation ()
```

501

7. Override is a standard annotation to signal to the Java runtime that one method overrides another.

8. Deprecated can also be considered a meta-annotation because it is applicable to other annotations.

9. SuppressWarnings is a single value annotation. Deprecated and Override, on the other hand, are both markers annotations.

10. It is not possible to use the SuppressWarnings, Deprecated and Override simultaneously on a single class

Exercise 16.c)

Create a marker annotation for classes, called Short, which must be used to mark classes that have no more than three methods. This annotation must belong to a package named metadata. Then create two classes ShortClass and LargeClass, the first with only one method, and the other with four methods. Let's annotate both classes with Short, and let them belong to a data package. Also create an exception that we will call AnnotationException to trigger when a class does not comply with the annotation specification, which must belong to the excs package.

Exercise 16.d)

Create an InteractiveChecker class that contains a main() method that checks if a class specified at runtime through the use of a Scanner class and annotated with Short, satisfies the requirement we specified in exercise 16.c. In other words, it must be possible to specify a class, press the **Enter** key and the program must print if the verification was successful or an error message. This class must belong to a package called test.

Exercise 16.e)

Create a single value annotation named Specification, which must be used to mark classes that have a precise number of instance variables. Also make this annotation belong to the metadata package. Also use this annotation for the ShortClass and LargeClass classes, which will declare instance variables. In particular ShortClass will declare an encapsulated instance variable, while LargeClass will declare two non-encapsulated instance variables.

Exercise 16.f)

Modify the InteractiveChecker class to verify the correctness of using the Specification annotation as done for the Short annotation.

502

Exercise 16.g)

Create a complete annotation called Bean, which must be used to mark classes that have a constructor without parameters, encapsulated variables, a number of methods not greater than a number to be specified, and a number of variables no less than a number to be specified. Also make this annotation belong to the metadata package. Also use this annotation to mark the ShortClass and LargeClass classes.

Exercise 16.h)

Modify the InteractiveChecker class to verify the correctness of using the Bean annotation as done for the Short annotation and the Specification annotation.

Exercise 16.i)

Which of the following statements are correct?

1. The Override annotation is useful only in the compilation phase.

2. An Override annotation that annotates a method always precedes any modifier of the method.

3. The Override annotation belongs to the java.lang package.

4. The Override annotation can only annotate methods.

5. The Override annotation is a marker annotation.

Exercise 16.l)

Given the following hierarchy:

```
public interface Player {
    default void play() {}
}

public class Child implements Player {
    /*INSERT CODE HERE*/
}
```

What can you insert instead of the comment /*INSERT CODE HERE*/ among the following statements?

1. `@Override void play() {}`

2. `@Override public void play() {}`

3. `@Override public boolean equals(Object o) {return false;}`

4. `@Override public int hashcode() {return 1;}`

5. `@Override public String toString() {return "";}`

Exercise 16.m)

Which of the following statements are correct?

1. The `Deprecated` annotation should be used instead of the javadoc `@Deprecated` tag.

2. A `Deprecated` annotation is a complete annotation type, and declares two elements: since, and `forRemoval`.

3. The `Deprecated` annotation belongs to the `java.lang.annotation` package.

4. The `Deprecated` annotation is a meta-annotation.

Exercise 16.n)

Which of the following statements are correct?

1. The `FunctionalInterface` annotation is useful only in the compilation phase.

2. A `FunctionalInterface` annotation should only annotate interfaces that have a single method (which is called method SAM).

3. The use of the `FunctionalInterface` annotation is mandatory if the annotated interface has a single method.

4. The `FunctionalInterface` annotation is a marker annotation.

5. The use of the `FunctionalInterface` annotation surely implies the use of the `@Override` annotation.

Exercise 16.o)

If we compile the following class:

```
import java.util.*;
```

504

```
public class Exercise16O {

    List objects;

    public Exercise16O() {
        objects = new ArrayList();

    }

    public void remove(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

we will get the following output which indicates three warnings.

```
Exercise16O.java:4: warning: [rawtypes] found raw type: List
    List objects;
    ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
Exercise16O.java:7: warning: [rawtypes] found raw type: ArrayList
        objects = new ArrayList();
                      ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
Exercise16O.java:12: warning: [rawtypes] found raw type: Iterator
        Iterator iterator = objects.iterator();
        ^
  missing type arguments for generic class Iterator<E>
  where E is a type-variable:
    E extends Object declared in interface Iterator
3 warnings
```

You can add an annotation to get:

1. Only two warnings;

2. Just a warning;

3. No warnings.

Write the three versions of the class that accomplish what is required.

Exercise 16.p)

Which of the following statements are correct?

1. The Native annotation can be used to interface Java with other languages.

2. The term "native language" means Java language.

3. The Native annotation belongs to the java.lang package.

4. The Native annotation can only annotate constants.

5. The Native annotation is a marker annotation.

Exercise 16.q)

Which of the following statements are correct?

1. The Target annotation is a meta-annotation that annotate itself.

2. The Target annotation is an ordinary annotation type. In fact, it can specify various parameters.

3. ElementType is an interface that defines the various Java programming elements to which annotations can be applied.

4. The Target annotation can also mark annotations intended to annotate local variables and type uses (such as when defining a cast, or invoking a constructor).

5. The Target annotation can also mark annotations intended to annotate import instructions.

Exercise 16.r)

Which of the following statements are correct?

1. The Retention annotation is annotated in turn by Target.

2. The Target annotation is annotated in turn by Retention.

3. The Documented annotation is annotated in turn by Target.

4. The Target annotation is annotated in turn by Documented.

5. The Retention annotation is annotated in turn by Documented.

6. The Documented annotation is annotated in turn by Retention.

506

Exercise 16.s)

Which of the following statements are correct?

1. With the `Retention` annotation we decide whether the annotated annotation should or should not be kept within the compiled class.

2. `RetentionPolicy` is an enumeration that declares only two elements: `SOURCE` and `CLASS`.

3. The `Retention` annotation and the `RetentionPolicy` enumeration belong to the `java.lang.annotation` package.

4. The `Retention` annotation can only annotate annotations.

5. The `Retention` annotation is an ordinary annotation type.

Exercise 16.t)

Which of the following statements are correct?

1. The `Documented` annotation allows you to include the `Documented` annotation within the javadoc documentation.

2. A `Documented` annotation is annotated by itself.

3. The `Documented` annotation is inherited by default in the subclasses.

4. The `Documented` annotation can only mark annotations that annotate classes.

5. The `Documented` annotation is a marker annotation.

Exercise 16.u)

Given the following code:

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface Annotation16U {

}
```

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface DifferentAnnotation {

}

@Annotation16U
public interface Interface16U {

}

@DifferentAnnotation
public class Exercise16U implements Interface16U {

}
```

What is the output of the following program?

```
import java.lang.reflect.*;
import java.util.*;
import java.lang.annotation.*;
public class AnnotationsReflection {
    public static void main(String[] args) throws Exception {
        Annotation[] dcs=Exercise16U.class.getAnnotations();
        for (Annotation dc : dcs) {
            System.out.println(dc);
        }
    }
}
```

Exercise 16.v)

Which of the following statements are correct?

1. The Repeatable annotation can annotate only annotations, in order to make their use repeatable.

2. When using a annotation annotated correctly with Repeatable, the JVM creates an object for us on the fly.

3. The Repeatable annotation must be a single value and return an array.

4. To declare a Repeatable annotation you must also create another auxiliary annotation.

508

5. An annotation marked as `Repeatable` can annotate the same programming element several times.

Given the following code:

```
//...
@Check(">=0")
@Check("<100")
private int a;
//...
```

Create the `Check` annotation.

Chapter 16
Exercise Solutions

Annotation Types

Solution 16.a) Annotations, Declarations and Usage, True or False:

1. **False**, is an annotation type.

2. **False**, is an annotation type.

3. **True**.

4. **False**, an element of an annotation cannot have as a type void.

5. **False**, an annotation method cannot have input parameters.

6. **True**.

7. **False**, in fact it is legal both the code of the m() method, and to declare public before the annotation (but obviously it is a modifier of the method). It is not valid, however, to pass the value MyAnnotation.MyEnum.TRUE as an input to the annotation without specifying a syntax of the type key = value.

8. **False**, in fact, the syntax:

   ```
   return @MyAnnotation.myEnum;
   ```

   is not valid. An annotation cannot be used as if it were a class with public static variables.

9. **False**, in fact, it is not a single value annotation, because its only element is not called `value()`.

10. **True**.

Solution 16.b) Annotations API, True or False:

1. **True**, in fact, if you do not specify with the meta-annotation `Target` what are the elements to which the annotation is applicable, the annotation will be by default applicable to any element except parameter types and type uses.

2. **False**, the value `ElementType.SOURCE` doesn't exist.

3. **False**, the value `ElementType.@INTERFACE` doesn't exist.

4. **False**, it is also not applicable to methods due to the `Target` value, which is `ElementType.ANNOTATION_TYPE`.

5. **False**, in fact it cannot be applied to a class if it is annotated with `@Target (ElementType.METHOD)`.

6. **True**.

7. **False**, at compile-time, not at runtime.

8. **True**.

9. **True**.

10. **True**, `Override` is not applicable to classes.

Solution 16.c)

The code of the `Short` annotation should be similar to the following:

```
package metadata;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

512

```
@Inherited
public @interface Short {

}
```

Note that for our purposes we had to specify the value of `Retention` to `RetentionPolicy.RUNTIME`. The class `ShortClass` could instead be modified in the following way:

```
package data;

import metadata.Short;

@Short
public class ShortClass {

    public void method1() {
        System.out.println("method1");
    }
}
```

The `LargeClass` class instead, could be similar to the following:

```
package data;

import metadata.Short;

@Short
public class LargeClass {
    public void method1() {
        System.out.println("method1");
    }

    public void method2() {
        System.out.println("method2");
    }

    public void method3() {
        System.out.println("method3");
    }

    public void method4() {
        System.out.println("method4");
    }
}
```

Finally, here is the requested exception:

```
package excs;

public class AnnotationException extends Exception {
```

513

```
    public AnnotationException(String msg) {
        super(msg);
    }

    @Override
    public String toString() {
        return "AnnotationException{" + getMessage() + "}";
    }
}
```

Solution 16.d)

The required checker code could be as follows:

```
package test;

import excs.AnnotationException;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.util.Scanner;
import metadata.Short;

public class InteractiveChecker {

    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        String string = "";
        System.out.println("Type the name of a java file present in the "
            + "current folder and type enter, or write \"end\" "
            + "to end the program");
        while (!(string = scanner.next()).equals("end")) {
            System.out.println("You typed " + string.toUpperCase() + "!");
            try {
                checkClass(string);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
        System.out.println("Program terminated!");
    }

    private static void checkClass(String string) throws Exception {
        Class objectClass = Class.forName(string);
        try {
            System.out.println("Start checking @Short annotation for "
                + string);
            Annotation shortAnnotation = objectClass.getAnnotation(Short.class);
            if (shortAnnotation != null) {
```

```
                    Method[] methods = objectClass.getDeclaredMethods();
                    final int methodsNumber = methods.length;
                    if (methodsNumber > 3) {
                        throw new AnnotationException("There are " + methodsNumber
                          + " methods in the class " + string);
                    }
                    System.out.println("Class "+string+" valid!\nmethods list:");
                    for (Method method : methods) {
                        System.out.println(method);
                    }
                } else {
                    System.out.println("This class is not annotated with @Short");
                }
            } finally {
                System.out.println("End of Short annotation check for " + string);
            }
        }
    }
}
```

Just look the checkClass() method, whose code is still very intuitive.

The following output is produced by first specifying the two classes actually annotated with @Short, then one that is not annotated (the AnnotationException class) and finally a non-existent class:

```
Type the name of a java file present in the current folder and type
 enter, or write "end" to end the program
data.ShortClass
You typed DATA.SHORTCLASS!
Start checking @Short annotation for data.ShortClass
Class data.ShortClass valid!
methods list:
public java.lang.String data.ShortClass.getVariable()
public void data.ShortClass.method1()
public void data.ShortClass.setVariable(java.lang.String)
End of Short annotation check for data.ShortClass

You typed DATA.LARGECLASS!
Start checking @Short annotation for data.LargeClass
End of Short annotation check for data.LargeClass
AnnotationException{There are 4 methods in the class data.LargeClass}
    at test.InteractiveChecker.checkShort(InteractiveChecker.java:47)
    at test.InteractiveChecker.checkClass(InteractiveChecker.java:34)
    at test.InteractiveChecker.main(InteractiveChecker.java:24)

excs.AnnotationException
You typed EXCS.ANNOTATIONEXCEPTION!
Start checking @Short annotation for excs.AnnotationException
This class is not annotated with @Short
End of Short annotation check for excs.AnnotationException

NonExistentClass
You typed NONEXISTENTCLASS!
```

515

```
java.lang.ClassNotFoundException: NonExistentClass
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(
 BuiltinClassLoader.java:604)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.
 loadClass(ClassLoaders.java:178)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
    at java.base/java.lang.Class.forName0(Native Method)
    at java.base/java.lang.Class.forName(Class.java:333)
    at test.InteractiveChecker.checkClass(InteractiveChecker.java:33)
    at test.InteractiveChecker.main(InteractiveChecker.java:24)
```

Solution 16.e)

The code of the Specification annotation should be the following:

```
package metadata;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface Specification {
    int value();
}
```

The code of the classes ShortClass and LargeClass could be modified as follows:

```
package data;

import metadata.Short;
import metadata.Specification;

@Short
@Specification(1)
public class ShortClass {

    public void method1() {
        System.out.println("method1");
    }
    private String variable;

    public String getVariable() {
```

516

```
        return variable;
    }

    public void setVariable(String variable) {
        this.variable = variable;
    }
}
```

and:

```
package data;

import metadata.Short;
import metadata.Specification;

@Short
@Specification(3)
public class LargeClass {

    public String variable1;
    public String variable2;

    public void method1() {
        System.out.println("method1");
    }

    public void method2() {
        System.out.println("method2");
    }

    public void method3() {
        System.out.println("method3");
    }

    public void method4() {
        System.out.println("method4");
    }
}
```

Solution 16.f)

Let's modify the code of our checker, extracting as a method the verification code of the Short annotation, and commenting out its call. We create an equivalent method to verify the Specific annotation. Below only the modified code is reported:

```
    private static void checkClass(String string) throws Exception {
        Class objectClass = Class.forName(string);
        //checkShort(string, objectClass);
```

```
            checkSpecification(string, objectClass);
        }

    private static void checkSpecification(String string, Class objectClass)
        throws AnnotationException {
        try {
            System.out.println("Start check @Specification annotation for "
               + string);
            Specification specification =
               (Specification) objectClass.getAnnotation(Specification.class);
            if (specification != null) {
                int variablesNumberFromSpecification = specification.value();
                Field[] fields = objectClass.getDeclaredFields();
                final int variablesNumber = fields.length;
                if (variablesNumber != variablesNumberFromSpecification) {
                    throw new AnnotationException("There are "
                       + variablesNumber
                       + " variables in the class " + string
                       + " but they should be " +
                       variablesNumberFromSpecification);
                }
                System.out.println("Class "+string+" valid!\nvariables list:");
                for (Field field : fields) {
                    System.out.println(field);
                }
            } else {
                System.out.println("This class is not annotated with "
                   + @Specification");
            }
        } finally {
            System.out.println("End check @Specification annotation for "
               + string);
        }
    }
```

Here is an example of output:

```
Type the name of a java file present in the current folder and type enter, or write "end"
 to end the program
data.ShortClass
You typed DATA.SHORTCLASS!
Start check @Specification annotation for data.ShortClass
Class data.ShortClass valid!
variables list:
private java.lang.String data.ShortClass.variable
End check @Specification annotation for data.ShortClass

data.LargeClass
You typed DATA.LARGECLASS!
Start check @Specification annotation for data.LargeClass
End check @Specification annotation for data.LargeClass
```

518

```
AnnotationException{There are 2 variables in the class data.LargeClass
 but they should be 3}
    at test.InteractiveChecker.checkSpecification(
 InteractiveChecker.java:72)
    at test.InteractiveChecker.checkClass(InteractiveChecker.java:35)
    at test.InteractiveChecker.main(InteractiveChecker.java:24)
```

Solution 16.g)

The code of the Bean annotation could be the following:

```
package metadata;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface Bean {
    int methodsMaxNumber();
    int variablesMinNumber();
}
```

The list of classes ShortClass and LargeClass could be modified as follows:

```
package data;

import metadata.Bean;
import metadata.Short;
import metadata.Specification;

@Short
@Specification(1)
@Bean(methodsMaxNumber = 10, variablesMinNumber = 1)
public class ShortClass {

    public void method1() {
        System.out.println("method1");
    }
    private String variable;

    public String getVariable() {
        return variable;
    }
```

```
     public void setVariable(String variable) {
         this.variable = variable;
     }
 }
```

and:

```
package data;

import metadata.Bean;
import metadata.Short;
import metadata.Specification;

@Short
@Specification(3)
@Bean(methodsMaxNumber = 10, variablesMinNumber = 1)
public class LargeClass {

    public String variable1;
    public String variable2;

    public void method1() {
        System.out.println("method1");
    }

    public void method2() {
        System.out.println("method2");
    }

    public void method3() {
        System.out.println("method3");
    }

    public void method4() {
        System.out.println("method4");
    }
}
```

Solution 16.h)

Let's modify the code of our checker and, as done for the solution of Exercise 16.f, let's create a method to verify the Bean annotation. Below there is only the modified code:

```
private static void checkClass(String string) throws Exception {
        Class objectClass = Class.forName(string);
//        checkShort(string, objectClass);
//        checkSpecification(string, objectClass);
        checkBean(string, objectClass);
    }
```

520

```java
    private static void checkBean(String string, Class objectClass) throws
      AnnotationException, NoSuchMethodException {
        try {
            System.out.println("Start check @Bean annotation per " + string);
            Bean bean = (Bean) objectClass.getAnnotation(Bean.class);
            if (bean != null) {
                checkVariablesNumber(bean, objectClass, string);
                checkMethodsNumber(bean, objectClass, string);
                checkNoArgumentsConstructor(objectClass, string);
                checkEncapsulation(objectClass, string);
            } else {
                System.out.println("This class is not annotated with @Bean");
            }
        } finally {
            System.out.println("End check @Bean annotation for " + string);
        }
    }

    private static void checkVariablesNumber(Bean bean, Class objectClass,
      String string) throws AnnotationException {
        int variablesMinNumber = bean.variablesMinNumber();
        Field[] fields = objectClass.getDeclaredFields();
        final int variablesNumber = fields.length;
        if (variablesNumber < variablesMinNumber) {
            throw new AnnotationException("There are " + variablesNumber
              + " variables in the class " + string
              + " but they should be at least " + variablesMinNumber);
        }
        System.out.println("Class " + string +
          ": variables number ok!\nvariables list:");
        for (Field field : fields) {
            System.out.println(field);
        }
    }

    private static void checkMethodsNumber(Bean bean, Class objectClass,
      String string) throws AnnotationException, NoSuchMethodException {
        int methodsMaxNumber = bean.methodsMaxNumber();
        Method[] methods = objectClass.getDeclaredMethods();
        final int methodsNumber = methods.length;
        if (methodsNumber > methodsMaxNumber) {
            throw new AnnotationException("There are " + methodsNumber
              + " methods in the class " + string
              + " but thau should be a maximum of " + methodsMaxNumber);
        }
        System.out.println("Class " + string
          + ": methods number ok!\nmethods list:");
        for (Method method : methods) {
            System.out.println(method);
```

521

```
        }
    }

    private static void checkNoArgumentsConstructor(Class objectClass,
      String string) throws AnnotationException, NoSuchMethodException {
        Constructor constructor = objectClass.getConstructor();
        if (constructor == null) {
            throw new AnnotationException(
              "No constructor without parameters!");
        }
        System.out.println("Class " + string
          + " no-arguments constructor present!:");
        System.out.println(constructor);
    }

    private static void checkEncapsulation(Class objectClass, String string)
      throws AnnotationException, NoSuchMethodException {
        Field[] fields = objectClass.getDeclaredFields();
        for (Field field : fields) {
            final String variableName = field.getName();
            final Class<?> type = field.getType();
            final Method setMethod = objectClass.getDeclaredMethod("set" +
                capitalize(variableName), type);
            final Method getMethod = objectClass.getDeclaredMethod("get" +
                capitalize(variableName));
            if (setMethod == null || getMethod == null ||
                !getMethod.getReturnType().equals(type)) {
                throw new AnnotationException("Variable " + variableName +
                  " not properly encapsulated in the class" + string);
            }
        }
        System.out.println("Class " + string + ": encapsulation ok!");

    }

    private static String capitalize(String string) {
        return string.substring(0, 1).toUpperCase() + string.substring(1);
    }
```

Ecco un esempio di output:

```
Type the name of a java file present in the current folder and type enter, or write "end"
 to end the program
data.ShortClass
You typed DATA.SHORTCLASS!
Start check @Bean annotation per data.ShortClass
Class data.ShortClass: variables number ok!
variables list:
private java.lang.String data.ShortClass.variable
Class data.ShortClass: methods number ok!
```

```
methods list:
public void data.ShortClass.setVariable(java.lang.String)
public void data.ShortClass.method1()
public java.lang.String data.ShortClass.getVariable()
Class data.ShortClass no-arguments constructor present!:
public data.ShortClass()
Class data.ShortClass: encapsulation ok!
End check @Bean annotation for data.ShortClass

data.LargeClass
You typed DATA.LARGECLASS!
Start check @Bean annotation per data.LargeClass
Class data.LargeClass: variables number ok!
variables list:
public java.lang.String data.LargeClass.variable1
public java.lang.String data.LargeClass.variable2
Class data.LargeClass: methods number ok!
methods list:
public void data.LargeClass.method1()
public void data.LargeClass.method2()
public void data.LargeClass.method3()
public void data.LargeClass.method4()
Class data.LargeClass no-arguments constructor present!:
public data.LargeClass()
End check @Bean annotation for data.LargeClass
java.lang.NoSuchMethodException: data.LargeClass.setVariable1(java.lang.String)
    at java.base/java.lang.Class.getDeclaredMethod(Class.java:2476)
    at test.InteractiveChecker.checkEncapsulation(
 InteractiveChecker.java:152)
    at test.InteractiveChecker.checkBean(InteractiveChecker.java:97)
    at test.InteractiveChecker.checkClass(InteractiveChecker.java:36)
    at test.InteractiveChecker.main(InteractiveChecker.java:24)
```

Solution 16.i)

All statements on the `Override` annotation are correct.

Solution 16.l)

Only statement 1 is incorrect as, by not making the `public` modifier explicit, it is making the method that applies the override, less accessible than the inherited original. The latter in fact, being a method declared in an interface, is implicitly `public`. In short, the rule mentioned in section 8.2.3.1 (third rule) has been violated.

In fact, the possible output would be the following:

```
Child.java:2: error: play() in Child cannot implement play() in Player
    /*INSERT CODE HERE*/@Override void play() {}
                                    ^
  attempting to assign weaker access privileges; was public
1 error
```

Solution 16.m)

Only the second statement is correct. The first one is not correct because the annotation and the javadoc tag should be used simultaneously. The third is also incorrect because the annotation belongs to java.lang. Finally, the number 4 is wrong because this annotation can annotate different elements of the Java programming, but not other annotations.

Solution 16.n)

The correct statements are 1, 2 and 4. The third is incorrect since it is possible to declare a functional interface without having to annotate it.

The number 5 is incorrect instead, because as FunctionalInterface also the annotation @Override is always optional.

Solution 16.o)

To get only two warnings there are more solutions: add an annotation to suppress the annotations either on the constructor, or on the instance variable objects, or on the remove() method. For example, we choose the instance variable objects:

```java
import java.util.*;

public class Exercise16O {
    @SuppressWarnings({"rawtypes"})
    List objects;

    public Exercise16O(){
        objects = new ArrayList();
    }
    public void remove(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

That produce the following output:

```
Exercise16O.java:8: warning: [rawtypes] found raw type: ArrayList
        objects = new ArrayList();
                      ^
  missing type arguments for generic class ArrayList<E>
```

524

```
   where E is a type-variable:
     E extends Object declared in class ArrayList

Exercise16O.java:12: warning: [rawtypes] found raw type: Iterator
        Iterator iterator = objects.iterator();
        ^
  missing type arguments for generic class Iterator<E>
  where E is a type-variable:
    E extends Object declared in interface Iterator
2 warnings
```

It's easy to imagine how to get just one warning. For example, annotating the constructor and `remove()` method:

```java
import java.util.*;

public class Exercise16O {

    List objects;

    @SuppressWarnings({"rawtypes"})
    public Exercise16O(){
        objects = new ArrayList();
    }

    @SuppressWarnings({"rawtypes"})
    public void remove(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

In this case we will get the following output:

```
Exercise16O.java:5: warning: [rawtypes] found raw type: List
    List objects;
    ^
  missing type arguments for generic class List<E>
  where E is a type-variable:
    E extends Object declared in interface List
1 warning
```

Finally, we could annotate the instance variable and to not get any warning, but it is more convenient to note the entire class:

```
import java.util.*;

@SuppressWarnings({"rawtypes"})
public class Exercise16O {

    List objects;

    public Exercise16O() {
        objects = new ArrayList();
    }

    public void remove(Object object) {
        Iterator iterator = objects.iterator();
        if (iterator.hasNext()) {
            Object item = iterator.next();
            if (object.toString().equals(item.toString())) {
                iterator.remove();
            }
        }
    }
}
```

that does not produce compilation warnings.

Solution 16.p)

The correct statements are 1, 4 and 5. The term "native language" usually refers to the language used by the platform on which the program runs, that is the operating system, which usually coincides with C/C ++, so the number statement 2 is incorrect.
The Native annotation belongs to the java.lang.annotation package, so the statement number 3 is also incorrect.

Solution 16.q)

The correct statements are 1 and 4. The number 2 is incorrect because Target is a single value annotation of ElementType array.
The number 3 is not correct because ElementType is not an interface but an enumeration.
The number 5 is not correct because there are situations not contemplated by interfaces annotated with Target specifying TYPE_USE as ELEMENT_TYPE, and among these there is precisely the case of the import as we have specified in the last lines of the section 16.2.1.1.

Solution 16.r)

All statements are true.

526

Solution 16.s)

The correct statements are 1, 3 and 4.

The number 2 is incorrect because `RetentionPolicy` also declares a third element (RUNTIME). The number 5 is not correct because `Target` is a single value annotation of type `RetentionPolicy`.

Solution 16.t)

The correct statements are 2 and 5.

The number 1 is incorrect because the `Documented` annotation is an annotation that marks annotations reported within the generated javadoc documentation.

The numbers 3 and 4 are simply invented.

Solution 16.u)

The program that reads the `Exercise16U` annotations, will print only the annotation that has been declared for the class itself, but will not inherit the annotation from the `Interface16U` interface. In fact, the `Inherited` annotation works only on classes and not on interfaces.

Here is the output:

```
@DifferentAnnotation()
```

Solution 16.v)

The correct statements are 1, 2, 4 and 5.

The number 3 is not correct because the single value annotation that must return an array, the one we have called the *container annotation*, is the auxiliary annotation discussed in the statement 4.

Solution 16.z)

We could create the following annotations: the `Check` annotation, and its `Checks` container annotation:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Inherited
@Documented
```

527

```
@Repeatable(Checks.class)
public @interface Check {
    String value();
}
```

and:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Inherited
@Documented
public @interface Checks {
    Check [] value();
}
```

528

Chapter 17
Exercises

Lambda Expressions

Lambda expressions and method references are not simple topics, but are really important. The following exercises should allow the reader to better understand the topics covered in chapter 17. Also, several exercises have been introduced that support the Oracle certification.

Exercise 17.a) Lambda Expressions and Method References:

1. With a lambda expression it is possible to do everything that can be done with an anonymous class.

2. A lambda expression can use instance variables of the class in which it is declared in a thread-safe manner.

3. A lambda expression can use the instance variables of the class in which it is defined only if declared `final`.

4. The following lambda expression is valid:

```
Consumer <String> c = ((x)->System.out.println(x));
```

5. The following lambda expression is valid:

```
(x, y) -> System.out.println(x);
          System.out.println(y);
```

6. The following method reference expression is valid:

```
System.out::println()
```

7. The following statement is valid:

```
System.out.println(Math::random)
```

8. The following statement is valid:

```
System.out::println(Math::random)
```

9. The following statement is valid:

```
Supplier<Guitar> guitarSupplier = Guitar::new;
```

10. Assuming that the `Guitar` class has a constructor without parameters, then the following code is valid:

```
Guitarist guitar = new Guitarist();
guitar.playGuitar(Guitar::new);
```

Exercise 17.b)

Create a `ComparatorsTest` class containing a `main()` method that declares an array of strings (at least two of which have the same length). Create `Comparator` objects with reference to methods, to sort the strings according to the following criteria:

- ▪ by length (from the longest string to the shortest string);

- ▪ by length in reverse (from the shortest string to the longeststring);

- ▪ by alphabetical order (however, use a lambda expression or a method reference, even if there is no need for it);

- ▪ by reverse alphabetical order;

- ▪ by length, but in the case of two strings with the same length by alphabetical order.

Sort the array using the `sort()` method of the `Arrays` class, which takes the declared array as the first parameter and a `Comparator` object as the second parameter. After each sort print the result. Create this class in a package, for example `com.claudiodesio.lambda.test`.

Exercise 17.c)

Create a `City` class, which abstracts the concept of city, which declares the `name` string, and the `stateCapital` and `onTheSea` booleans, as encapsulated variables.

530

> **Remember that getter methods for booleans usually use the "is" prefix instead of "get". So the `getStateCapital()` and `getOnTheSea()` methods should be written as `isOnTheSea()` and `isStateCapital()`.**

Also create any utility methods such as `toString()` (just to return the name) and a constructor. This class will belong to a package such as `com.claudiodesio.lambda.dati`.
Create an `Exercise17C` class with a `main()` method, which prints:

- the list of cities on the sea (for example `Cities on the sea: [Siracusa, Napoli, Pescara, Taranto]`)

- the list of state capital cities (for example, `State  Capitals: [Milano, Potenza, Perugia, Napoli]`)

using the reference methods and the `Predicate` interface.
This class will belong to a package, for example `com.claudiodesio.lambda.test`.

Exercise 17.d)

After doing the previous exercise, create an `Exercise17D` alternative to `Exercise17C`, which performs the same operations using lambda expressions.

Exercise 17.e)

Create the `Exercise17E` class by editing the `Exercise17C` class by adding a `printDetails()` method that prints the list of cities, even with the information defined by the `stateCapital` and `onTheSea` variables. For example, the output might be similar to the following:

```
Milano is a capital state,
Pescara is on the sea,
Napoli is a capital state, is on the sea,
...
```

To get this result, you don't have to change the `City` class, but use a lambda expression in `Exercise17E`, taking advantage of one of the standard functional interfaces, which one?

Exercise 17.f)

Repeat Exercise 11.h, using a lambda expression instead of the anonymous class. For convenience, we report code of the Exercise 10.h below.

Suppose we want to write a program and want to re-use the following `Person` class, inherited from a program already written and not editable:

```java
public class Person {

    private String name;
    private String surname;
    private String birthDate;
    private String occupation;
    private String address;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public Person(String name, String surname, String birthDate,
            String occupation, String address) {
        this.name = name;
        this.surname = surname;
        this.birthDate = birthDate;
        this.occupation = occupation;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public String getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(String birthDate) {
        this.birthDate = birthDate;
    }
```

```
    public String getOccupation() {
        return occupation;
    }

    public void setOccupation(String occupation) {
        this.occupation = occupation;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Person{" + "name=" + name + ", surname=" + surname + "}";
    }
}
```

Unfortunately, in our context, we would need to redefine the toString() method so that it not only prints information on the person's first and last name, but also the date of birth, address and occupation. As already mentioned, however, the class is already in use and cannot be changed. In particular, our requirement is that the toString() method returns the following string:

```
Name:                 Arjen Anthony
Surname:              Lucassen
Occupation:     Composer
Birth Date      03/04/1960
Address:              Holland
```

Then create a PersonTest class that redefines the toString() method of the Person class and prints the above output.

Exercise 17.g)

Consider the Observation class of the solution of the Exercise 15.c, which we report below for convenience:

```
package com.claudiodesio.observatory.test;

import com.claudiodesio.observatory.data.Participant;
import com.claudiodesio.observatory.data.Telescope;
```

```
public class Observation {

    public static void main(String args[]) {
        Telescope telescope = new Telescope();
        Participant[] participants = getParticipants(telescope);
        for (Participant participant : participants) {
            participant.start();
        }
    }

    private static Participant[] getParticipants(Telescope telescope) {
        Participant[] participants = {
            new Participant("Ciro", telescope),
            new Participant("Gianluca", telescope),
            new Participant("Pierluigi", telescope),
            new Participant("Gigi", telescope),
            new Participant("Nicola", telescope),
            new Participant("Pino", telescope),
            new Participant("Maurizio", telescope),
            new Participant("Raffaele", telescope),
            new Participant("Fabio", telescope),
            new Participant("Vincenzo", telescope)};
        return participants;
    }
}
```

Is it possible to use a lambda expression to implement an override of the `run()` method of the `Participant` class, for one of the participants?

Exercise 17.h)

Given the following code:

```
new Thread(()->System.out.print("Java");System.out.print("Java");).start();
```

which of the following statements are correct?

1. Running this snippet with JShell prints the JavaJava string.

2. Running this snippet with JShell prints the Java string.

3. This snippet will not pass the compilation.

4. This snippet will not throw an exception at runtime.

5. This snippet will produce a compilation warning.

Exercise 17.i)

Given the following code:

```
new Thread((int a)->{
  int b = 0;
  b = a/b;
}).start();
```

which of the following statements are correct?

1. The code does not compile because an `ArithmeticException` must be handled.

2. The code compiles and runs correctly.

3. The code compiles but will launch an `ArithmeticException` during execution.

4. The code will produce a compilation warning.

Exercise 17.l)

Suppose we have a lambda expression that uses a method that runs a checked exception without handling it. Which of the following statements is correct?

1. The code does not compile because we need to handle a checked exception in the code block of the lambda expression.

2. The code does not compile, but if it is possible, we can fix it by redefining the SAM method of the functional interface by handling the exception within the method itself.

3. The code quietly compiles.

4. Since we have to handle the exception, we lose the compiler's inference capabilities, which leads us to write less code.

Exercise 17.m)

Create the functional interface that can satisfy the following use of lambda expressions:

```
Operation operation1 = (double a, double b) ->  a + b;
Operation operation2 = (double a, double b) ->  a - b;
Operation operation3 = (double a, double b) ->  a / b;
Operation operation4 = (double a, double b) ->  a * b;
```

Exercise 17.n)

There is a standard functional interface, capable of replacing the `Operation` functional interface of the previous exercise?

> **Use the documentation to resolve the exercise.**

Exercise 17.o)

Rewrite exercise 17.b, replacing lambda expressions with method references.

Exercise 17.p)

What is the name of the functional interface that best fits as a "Factory"?

1. `Predicate`
2. `Factory`
3. `Function`
4. `Supplier`
5. `Consumer`

Exercise 17.q)

Select the correct statements.
A functional interface:

1. Must be annotated with the `FunctionalInterface` annotation.
2. Must declare a single method.
3. It must necessarily be implemented.
4. It cannot be extended by another functional interface.
5. It cannot be extended by another interface

Which of the following statements are correct about constructor references?

1. The syntax is ClassName::New.

2. The syntax is ClassName::new().

3. The syntax is ClassName::new.

4. A reference to a constructor can be assigned to a reference of a functional interface, whose SAM method returns void.

5. With a reference to a constructor, we can replace the implementation of a functional interface.

Create a Person class that declares the name and age variables and that implements the Comparable interface in such a way that the inherited compareTo() method, sort the objects by increasing age, and, in the case of persons with the same age, in alphabetical order (considering the name). Also override the toString() method.
Then, given the following class:

```java
import java.util.Arrays;

public class Exercise17S {
    public static void main(String args[]) {
        Person [] persons = {
            new Person("Antonio",21),
            new Person("Bruno",20),
            new Person("Giorgio",19),
            new Person("Martino",22),
            new Person("Daniele",21)
        };
        Arrays.sort(persons, /*INSERT CODE HERE*/);
        System.out.println(Arrays.toString(persons));
    }
}
```

enter the correct code instead of the comment /*INSERT CODE HERE*/, so as to generate the following output:

```
[Giorgio, Bruno, Antonio, Daniele, Martino]
```

Exercise 17.t)

Starting from the `Person` class of Exercise 17.s, which functional interface could be used instead of the following method?

```
boolean hasThisName(Person person, String name) {
    return name.equals(person.getName());
}
```

Exercise 17.u)

Starting from the `Person` class of Exercise 17.s, and considering the following class:

```
import java.util.Arrays;
import java.util.function.BiPredicate;

public class Exercise17U {

    public static void main(String args[]) {
        Person [] persons = {
            new Person("Antonio",21),
            new Person("Bruno",20),
            new Person("Giorgio",19),
            new Person("Martino",22),
            new Person("Daniele",21)
        };
        Person personWithNameThatStartsWithD =
            getPersonWithNameThatStartsWithD("D", persons,
            /*INSERT CODE HERE*/);
            System.out.println(personWithNameThatStartsWithD);
    }

    static Person getPersonWithNameThatStartsWithD(String firstCharacter,
      Person[] persons, BiPredicate<String, Person> biPredicate) {
        for(Person person : persons) {
            if (biPredicate.test(firstCharacter, person)) {
                return person;
            }
        }
        return null;
    }
}
```

Which lambda expression can be inserted in place of the comment `/*INSERT CODE HERE*/` to retrieve the first `Person` object that has a name that starts with the "D" from the array?

Exercise 17.v)

Which of the following statements are correct?

1. With lambda expressions the `this` reference directly refers to the class in which the expression is included.

2. For lambda expressions, the same rules apply as for anonymous classes.

3. The syntax of a lambda expression allows the use of other nested lambda expressions.

4. The syntax of a lambda expression allows the use of reference to nested methods.

5. The syntax of a method reference allows to use of other nested lambda expressions.

6. The syntax of a method reference allows the use of reference to nested methods.

Exercise 17.z)

Which of the following statements are correct?

1. The functional interface that declares two generic types as input parameters and returns another one, is called `BiFunction`.

2. The functional interface that does not declare generic types as input parameters, but returns another one, is called `Provider`.

3. The functional interface that does not declare generic types as input parameters, but returns another one, is called `Supplier`.

4. The functional interface that declares a generic type as an input parameter and returns another one, is called `Function`.

5. The functional interface that declares three generic types as input parameters and returns another one, is called `TriFunction`.

6. The functional interface that declares a generic type as an input parameter and returns the same type, is called `UnaryOperator`.

7. You can chain multiple `UnaryOperators` using the default `and()` method.

8. The functional interface that declares a generic type as an input parameter and returns nothing, is called `Consumer`.

Chapter 17
Exercise Solutions

Lambda Expressions

Solution 17.a) Lambda Expressions and Method References:

1. **False**, for example in an anonymous class we can also define instance variables.

2. **False**.

3. **False**, cfr. section 17.1.3.2.

4. **False**.

5. **False**, the curly braces surrounding the method instructions are missing.

6. **False**, the brackets next to the `println` method are not part of the syntax.

7. **False**.

8. **False**.

9. **True**.

10. **True**.

Solution 17.b)

The requested code could be similar to the following:

```
package com.claudiodesio.lambda.test;

import java.util.Arrays;
import java.util.Comparator;

public class ComparatorsTest {

    static String names[] = {"Clarissa", "Jem", "Top", "Ermeringildo",
      "Iamaca", "Tom", "Arlequin", "Francesca", "Cumbus", "Blue"};

    public static void main(String args[]) {
        Comparator<String> lenghtComparator = (first, second)
                -> -(Integer.compare(first.length(), second.length()));

        Comparator<String> reverseLengthComparator = (first, second)
                -> (Integer.compare(first.length(), second.length()));

        Comparator<String> reverseAlphabetComparator = (first, second)
                -> -(first.compareTo(second));

        Comparator<String> lengthAndReverseAlphabetComparator = (first, second)
                -> {
                    int result = -Integer.compare(first.length(),
                        second.length());
                    if (result == 0) {
                        result = first.compareTo(second);
                    }
                    return result;
                };
        Arrays.sort(names, lenghtComparator);
        System.out.println("Names sorted by length: " + Arrays.asList(names));

        Arrays.sort(names, reverseLengthComparator);
        System.out.println("Names sorted by length in reverse: " +
          Arrays.asList(names));

        Arrays.sort(names, String::compareTo);
        System.out.println("Names sorted by alphabetical order: " +
          Arrays.asList(names));

        Arrays.sort(names, reverseAlphabetComparator);
        System.out.println("Names sorted by reverse alphabetical order: " +
          Arrays.asList(names));

        Arrays.sort(names, lengthAndReverseAlphabetComparator);
        System.out.println("Names sorted by length in reverse and in " +
          "alphabetical order: " + Arrays.asList(names));
    }
}
```

542

Note that the `asList()` method of the `Arrays` class was used only to take advantage of the textual representation of the collection.
The output will be:

```
Names sorted by length: [Ermeringildo, Francesca, Clarissa, Arlequin, Iamaca, Cumbus,
 Blue, Jem, Top, Tom]
Names sorted by length in reverse: [Jem, Top, Tom, Blue, Iamaca, Cumbus, Clarissa,
 Arlequin, Francesca, Ermeringildo]
Names in alphabetical order: [Arlequin, Blue, Clarissa, Cumbus, Ermeringildo,
 Francesca, Iamaca, Jem, Tom, Top]
Names in reverse alphabetical order: [Top, Tom, Jem, Iamaca, Francesca, Ermeringildo,
 Cumbus, Clarissa, Blue, Arlequin]
Names sorted by length in reverse and in alphabetical order:
  [Ermeringildo, Francesca, Arlequin, Clarissa, Cumbus, Iamaca, Blue, Jem, Tom, Top]
```

Solution 17.c)

The code of the `City` class should be the following:

```java
package com.claudiodesio.lambda.data;

public class City {

    private String name;

    private boolean stateCapital;

    private boolean onTheSea;

    public City(String name, boolean stateCapital, boolean onTheSea) {
        this.name = name;
        this.stateCapital = stateCapital;
        this.onTheSea = onTheSea;
    }

    public boolean isOnTheSea() {
        return onTheSea;
    }

    public void setOnTheSea(boolean onTheSea) {
        this.onTheSea = onTheSea;
    }

    public boolean isStateCapital() {
        return stateCapital;
    }
```

```
    public void setStateCapital(boolean stateCapital) {
        this.stateCapital = stateCapital;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return getName();
    }
}
```

The code of the Exercise17C class instead, could be coded in the following way:

```
package com.claudiodesio.lambda.test;

import com.claudiodesio.lambda.data.City;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Predicate;

public class Exercise17C {

    public static void main(String args[]) {
        List<City> listOfCities = getCities();
        System.out.println("Cities on the sea: " +
          filterCities(listOfCities, Exercise17C::isOnTheSea));
        listOfCities = getCities();
        System.out.println("State capitals: " +
          filterCities(listOfCities, Exercise17C::isStateCapital));
        listOfCities = getCities();
    }

    public static List<City> filterCities(List<City> listOfCities,
      Predicate<City> p) {
        final Iterator<City> iterator = listOfCities.iterator();
        while (iterator.hasNext()) {
            City city = iterator.next();
            if (!p.test(city)) {
                iterator.remove();
            }
        }
        return listOfCities;
    }
```

```
        private static List<City> getCities() {
            List<City> city = new ArrayList<>();
            city.add(new City("Milano", true, false));
            city.add(new City("Rovigo", false, false));
            city.add(new City("Potenza", true, false));
            city.add(new City("Siracusa", false, true));
            city.add(new City("Perugia", true, false));
            city.add(new City("Napoli", true, true));
            city.add(new City("Pescara", false, true));
            city.add(new City("Taranto", false, true));
            city.add(new City("Siena", false, false));
            return city;
        }

        public static boolean isOnTheSea(City city) {
            return city.isOnTheSea();
        }

        public static boolean isStateCapital(City city) {
            return city.isStateCapital();
        }
    }
```

The output will be:

```
Cities on the sea: [Siracusa, Napoli, Pescara, Taranto]
State capitals: [Milano, Potenza, Perugia, Napoli]
```

Solution 17.d)

The code of the Exercise17D class could be the following:

```
package com.claudiodesio.lambda.test;

import com.claudiodesio.lambda.data.City;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Predicate;

public class Exercise17D {

    public static void main(String args[]) {
        List<City> listOfCities = getCities();
        System.out.println("Cities on the sea: " +
          filterCities(listOfCities, (city) -> city.isOnTheSea()));
        listOfCities = getCities();
        System.out.println("State capitals: " +
          filterCities(listOfCities,(city) -> city.isStateCapital()));
        listOfCities = getCities();
    }
```

```
      public static List<City> filterCities(List<City> listOfCities,
         Predicate<City> p) {
          final Iterator<City> iterator = listOfCities.iterator();
          while (iterator.hasNext()) {
              City city = iterator.next();
              if (!p.test(city)) {
                  iterator.remove();
              }
          }
          return listOfCities;
      }

      private static List<City> getCities() {
          List<City> city = new ArrayList<>();
          city.add(new City("Milano", true, false));
          city.add(new City("Rovigo", false, false));
          city.add(new City("Potenza", true, false));
          city.add(new City("Siracusa", false, true));
          city.add(new City("Perugia", true, false));
          city.add(new City("Napoli", true, true));
          city.add(new City("Pescara", false, true));
          city.add(new City("Taranto", false, true));
          city.add(new City("Siena", false, false));
          return city;
      }
  }
```

The output will remain the same as for the previous exercise:

```
Cities on the sea: [Siracusa, Napoli, Pescara, Taranto]
State capitals: [Milano, Potenza, Perugia, Napoli]
```

Solution 17.e)

The Exercise17E class could be the following:

```
package com.claudiodesio.lambda.test;

import com.claudiodesio.lambda.data.City;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

public class Exercise17E {

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

```java
        List<City> listOfCities = getCities();
        System.out.println("Cities on the sea: " +
          filterCities(listOfCities, Exercise17E::isOnTheSea));
        listOfCities = getCities();
        System.out.println("State capitals: " +
          filterCities(listOfCities, Exercise17E::isStateCapital));
        listOfCities = getCities();
        printDetails(listOfCities, (city) -> System.out.println(city.getName()
          + (city.isStateCapital() ? " is capital state," : "")
          + (city.isOnTheSea() ? " is on the sea," : "")));
    }

    public static List<City> filterCities(List<City> listOfCities,
      Predicate<City> p) {
        final Iterator<City> iterator = listOfCities.iterator();
        while (iterator.hasNext()) {
            City City = iterator.next();
            if (!p.test(City)) {
                iterator.remove();
            }
        }
        return listOfCities;
    }

    public static void printDetails(List<City> listOfCities, Consumer<City> c) {
        for (City city : listOfCities) {
            c.accept(city);
        }
    }

    private static List<City> getCities() {
        List<City> city = new ArrayList<>();
        city.add(new City("Milano", true, false));
        city.add(new City("Rovigo", false, false));
        city.add(new City("Potenza", true, false));
        city.add(new City("Siracusa", false, true));
        city.add(new City("Perugia", true, false));
        city.add(new City("Napoli", true, true));
        city.add(new City("Pescara", false, true));
        city.add(new City("Taranto", false, true));
        city.add(new City("Siena", false, false));
        return city;
    }

    public static boolean isOnTheSea(City city) {
        return city.isOnTheSea();
    }

    public static boolean isStateCapital(City city) {
        return city.isStateCapital();
    }
}
```

And so, the functional interface to use was the Consumer interface.

It was enough then to invoke this method using a lambda expression (and a couple of ternary operators) in the main() method:

```
printDetails(listOfCities, (city) -> System.out.println(city.getName()
    + (city.isStateCapital() ? " is capital state," : "")
    + (city.isOnTheSea() ? " is on the sea," : "")));
```

The output will be:

```
Cities on the sea: [Siracusa, Napoli, Pescara, Taranto]
State capitals: [Milano, Potenza, Perugia, Napoli]
Milano is capital state,
Rovigo
Potenza is capital state,
Siracusa is on the sea,
Perugia is capital state,
Napoli is capital state, is on the sea,
Pescara is on the sea,
Taranto is on the sea,
Siena
```

Solution 17.f)

It is impossible in this case to replace the anonymous class with a lambda expression! Recall that a lambda expression works by implementing functional interfaces (with only a single abstract method).

Solution 17.g)

Again, it is impossible to use a lambda expression. Instead it is possible to use an anonymous class:

```
package com.claudiodesio.observatory.test;

import com.claudiodesio.observatory.data.Participant;
import com.claudiodesio.observatory.data.Telescope;

public class Observation {

    public static void main(String args[]) {
        Telescope telescope = new Telescope();
        Participant[] participants = getParticipants(telescope);
        for (Participant participant : participants) {
            participant.start();
        }
    }
```

```
        private static Participant[] getParticipants(Telescope telescope) {
            Participant[] participants = {
                new Participant("Ciro", telescope),
                new Participant("Gianluca", telescope),
                new Participant("Pierluigi", telescope),
                new Participant("Gigi", telescope),
                new Participant("Nicola", telescope) {
                    @Override
                    public void run() {
                        System.out.println(getName() + " I'm ready!");
                        super.run();
                    }
                },
                new Participant("Pino", telescope),
                new Participant("Maurizio", telescope),
                new Participant("Raffaele", telescope),
                new Participant("Fabio", telescope),
                new Participant("Vincenzo", telescope)};
            return participants;
        }
    }
```

Solution 17.h)

Only the third statement is correct. The output of JShell is as follows:

```
|  Error:
|  ')' expected
|  new
 Thread(()->System.out.print("Java");System.out.print("Java");).start();
|
```

In fact, braces are missing around the code block. If there were:

```
  new Thread(()->{System.out.print("Java");System.out.print("Java");}).start();
```

then the first statement would have been the correct one:

```
jshell> JavaJava
```

Solution 17.i)

None of the statements is correct. In fact, the code will not compile because the constructor of a thread must pass an implementation of the run() method, which does not take input parameters as specified in the code. By running this snippet on JShell we will get the following output:

```
jshell> new Thread((int a)->{
   ...>   int b = 0;
   ...>   b = a/b;
   ...> }).start();
|  Error:
|  no suitable constructor found for Thread((int a)->{[...] b; })
|      constructor java.lang.Thread.Thread(java.lang.Runnable) is not
 applicable
|        (argument mismatch; incompatible parameter types in lambda
 expression)
|      constructor java.lang.Thread.Thread(java.lang.String) is not
 applicable
|        (argument mismatch; java.lang.String is not a functional
 interface)
|  new Thread((int a)->{
|  ^--------------------...
```

We note that ArithmeticException extends RuntimeException that is an unchecked exception. The compiler would therefore not have reported errors, and for this reason the first statement was certainly incorrect. The second and fourth statements are obviously false. The third statement would have been correct if the implementation of the run() method had been corrected, but it is not.

Solution 17.l)

The statements are all correct except the number 3.

Solution 17.m)

The solution is trivial:

```
public interface Operation {
    double operation(double x, double y);
}
```

Solution 17.n)

Yes, there is the DoubleBinaryOperator class that defines the method:

```
double applyAsDouble(double left, double right)
```

which coincides as parameters and return type with the operation() method of the Operation interface of the previous exercise (the name does not count).

Solution 17.o)

The solution could be the following:

```java
package com.claudiodesio.lambda.test;

import java.util.Arrays;
import java.util.Comparator;

public class Exercise17O {

    static String names[] = {"Clarissa","Jem","Top","Ermeringildo","Iamaca",
      "Tom","Arlequin","Francesca","Cumbus","Blue"
    };

    static int compareLength(String first, String second) {
        return -(Integer.compare(first.length(), second.length()));
    }

    static int compareReverseLength(String first, String second) {
        return (Integer.compare(first.length(), second.length()));
    }

    static int compareReverseAlphabet(String first, String second) {
        return -(first.compareTo(second));
    }

    static int compareLengthAndReverseAlphabet(String first, String second) {

        int result = -Integer.compare(first.length(), second.length());

        if (result == 0) {
            result = first.compareTo(second);
        }

        return result;
    }

    public static void main(String args[]) {

        Arrays.sort(names, Exercise17O::compareLength);
        System.out.println("Names sorted by length: " +
          Arrays.asList(names));

        Arrays.sort(names, Exercise17O::compareReverseLength);
        System.out.println("Names sorted by length in reverse: " +
          Arrays.asList(names));
```

```
        Arrays.sort(names, String::compareTo);
        System.out.println("Names in alphabetical order : " +
          Arrays.asList(names));

        Arrays.sort(names, Exercise17O::compareReverseAlphabet);
        System.out.println("Names in reverse alphabetical order " +
          Arrays.asList(names));

        Arrays.sort(names, Exercise17O::compareLengthAndReverseAlphabet);
        System.out.println(
          "Names sorted by length in reverse and in alphabetical order: "
          + Arrays.asList(names));
    }
}
```

Solution 17.p)

The exercise was really simple, the right answer is the number 4, `Supplier`, as the definition of its method is the following:

```
T get()
```

where `T` is a generic parameter of the interface.
Note that the `Factory` functional interface of answer 2 does not exist.

Solution 17.q)

None of the statements is correct.
Only the number 2 may not be clearly incorrect. In fact, a functional interface must declare a single abstract method, but it is also possible to declare default and static methods, both public and private.

Solution 17.r)

The correct statements are the number 3 (which excludes the correctness of 1 and 2) and the number 5 (which is consistent with the definition of lambda expressions). The statement number 4 is incorrect because the SAM method cannot return `void`, but must return the same type as the constructor.

Solution 17.s)

The `Person` class could be the following:

552

```
import java.util.*;

public class Person implements Comparable<Person> {

    private String name;

    private int age;

    public Person (String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Person otherPerson) {
        int result = Integer.valueOf(this.age).compareTo(
            Integer.valueOf(otherPerson.age));
        if (result == 0) {
            result = this.name.compareTo(otherPerson.name);
        }
        return result;
    }

    public String toString() {
        return name;
    }
}
```

Instead the Exercise17S class can be completed with the "reference to an instance method of a certain type" (which we studied in section 17.2.4) in the following way (the added code in bold):

```
import java.util.Arrays;
```

```
public class Exercise17S {
    public static void main(String args[]) {
        Person [] persons = {
            new Person("Antonio",21),
            new Person("Bruno",20),
            new Person("Giorgio",19),
            new Person("Martino",22),
            new Person("Daniele",21)
        };
        Arrays.sort(persons, Person::compareTo);
        System.out.println(Arrays.toString(persons));
    }
}
```

Solution 17.t)

The right answer is number 6: `BiPredicate`. In fact it defines the method:

```
boolean test(T t, U u)
```

Solution 17.u)

The solution could be the following (the added code is in bold):

```
import java.util.Arrays;
import java.util.function.BiPredicate;

public class Exercise17U {
    public static void main(String args[]) {
        Person [] persons = {
            new Person("Antonio",21),
            new Person("Bruno",20),
            new Person("Giorgio",19),
            new Person("Martino",22),
            new Person("Daniele",21)
        };
        Person personWithNameThatStartsWithD = getPersonWithNameThatStartsWithD(
            "D", persons, /*INSERT CODE HERE*/
            (String firstCharacter, Person person) ->
            person.getName().startsWith(firstCharacter));
        System.out.println(personWithNameThatStartsWithD);
    }

    static Person getPersonWithNameThatStartsWithD(String firstCharacter,
        Person[] persons, BiPredicate<String, Person> biPredicate) {
        for(Person person : persons) {
            if (biPredicate.test(firstCharacter, person)) {
```

```
                    return person;
                }
            }
        return null;
    }
}
```

Solution 17.v)

The correct answers are the numbers 1, 3 and 4. The number 2 is false, indeed the fact that lambda expressions do not inherit the complicated rules of anonymous classes is one of the advantages of using lambda expressions instead of anonymous classes. The numbers 5 and 6 are false because the syntax of a method reference does not include nested code, and therefore it is impossible to use other nested lambda expressions or other method references

Solution 17.z)

The correct answers are the numbers 1, 3, 4 and 7. The number 2 is false since the number 3 is correct.

The number 5 is false because `TriFunction` does not exist, but could easily be created. The number 6 is false because the `and()` method does not exist within `UnaryOperator` (if anything, it exists `andThen()`).

Chapter 18
Exercises

Collections Framework and Stream API

The Collections framework contains some of the most widely used implementations in Java. The documentation is fundamental, every good programmer should consult it assiduously. Being a very extensive library, there will always be a method or implementation that is right for us, which we have never used. The Stream API has further extended the horizons of application of the collections. We advise (as already done in the introduction) to comment on each single line implemented to better memorize the definitions and meaning of some statements.

Exercise 18.a) Framework Collections, True or False:

1. `Collection`, `Map`, `SortedMap`, `Set`, `List` and `SortedSet` are interfaces and cannot be instantiated.

2. A `Set` is an ordered collection of objects; a `List` does not allow duplicate elements and is ordering.

3. Maps cannot contain duplicate keys and each key can be associated with only one value.

4. There are several abstract implementations to customize in the framework such as `AbstractMap`.

5. A `HashMap` is faster than a `Hashtable` because it is not synchronized.

6. A HashMap is faster than a TreeMap but the latter, being an implementation of SortedMap, handles the sorting.

7. HashSet is faster than TreeSet but does not handle the sorting.

8. Iterator and Enumeration have the same role, but the latter allows during the iterations to also remove elements.

9. ArrayList has better performance than Vector because it is not synchronized, but both have mechanisms to optimize performance.

10. The Collections class is a Collection list.

Exercise 18.b) Stream API, True or False:

1. Stream is a class that implements Collection.

2. It is possible to iterate over a stream in both directions.

3. A pipeline consists of a source, optional aggregation methods and a terminal method.

4. The map() method is to be considered an aggregation method.

5. Optional is an interface that allows you to avoid having to deal with NullPointerException.

6. The reduction methods are aggregation operations.

7. The joining() method of Stream allows you to concatenate strings with string type separators.

8. The DoubleSummaryStatistics class is a particular stream.

9. The parallelStream() method returns a stream capable of using the Fork/Join algorithm to perform operations on the elements of a collection.

10. A stream can only be instantiated using a collection.

Exercise 18.c)

Create a collection that, if you add two equal elements, launch a custom exception (to be created as well). Finally also create a test class.

Exercise 18.d)

Create a map called `IncrementalMap`, which allows you to add more values (sorted as they added) to the same key. Also create a test class.

Exercise 18.e)

Taking advantage of the classes created in the previous exercises, create an `IncrementalMap` extension (let's call it `RobustIncrementalMap`) which, when adding an element whose key already exists, throws an exception as described in the exercise 18.c. Also create a test class.

Exercise 18.f)

After completing the previous exercise, consider the `City` class that was created in exercise 16.c and which we report below for convenience:

```java
package com.claudiodesio.data;

import java.util.Objects;

public class City {

    private String name;

    private boolean stateCapital;

    private boolean onTheSea;

    public City(String name, boolean stateCapital, boolean onTheSea) {
        this.name = name;
        this.stateCapital = stateCapital;
        this.onTheSea = onTheSea;
    }

    public boolean isOnTheSea() {
        return onTheSea;
    }

    public void setOnTheSea(boolean onTheSea) {
        this.onTheSea = onTheSea;
    }

    public boolean isStateCapital() {
        return stateCapital;
    }
```

```
    public void setStateCapital(boolean stateCapital) {
        this.stateCapital = stateCapital;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return getName();
    }
}
```

Add `equals()` and `hashcode()` methods. In a `StreamTest` cass, wth a stream, print all the "cities on the sea". With a second stream, print all the state capitals. With a third stream, print all the cities that end with the letter "a".

Exercise 18.g)

Regarding the enhanced for loop, which of the following statements are correct?

1. The enhanced `for` loop can in any case replace a `for` loop.

2. The enhanced `for` loop can be used with arrays and with classes that implement `Iterable`.

3. The enhanced `for` loop replaces the use of `Iterator`.

4. The enhanced `for` loop cannot correctly exploit `Iterator` methods.

5. In an enhanced `for` loop it is not possible loop backwards.

Exercise 18.h)

Given the following code:

```
import java.util.*;

public class Exercise18H {
```

```
    public static void main(String args[]) {
        Collection map = new HashMap(10);
        map.put(1,1);
        System.out.println(map);
    }
}
```

Which of the following statements is correct?

1. The code compiles with warnings, and when executed prints {1 = 1}.

2. The code compiles with warnings, but throws an exception during execution.

3. The code does not compile.

4. The code compiles correctly, and when executed prints {1 = 1}.

5. The code compiles with warnings, and when executed prints {1 = 1, null = null}.

6. The code compiles with warnings, and when executed prints {1 = 1, 0 = 0}.

Exercise 18.i)

Which of the following statements are correct?

1. The Iterable interface declares the forEach() method.

2. Iterator extends Iterable.

3. Iterator defines the forEachRemaining() method.

4. Collection implements the Iterable interface.

Exercise 18.l)

Which of the following statements are correct??

1. Collection is a superclass of List.

2. A Collection can be transformed into an array by invoking the toArray() method defined in the Arrays class.

3. An array can be transformed into a Collection using the toCollection() method of the Arrays class.

4. Collection defines the add() method.

Exercise 18.m)

Consider the following class:

```
import java.util.*;

public class Exercise18M {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<>(3);
        list.add("*");
        list.add("@");
        list.set(1, "$");
        ListIterator listIterator = list.listIterator();
        while(listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }
        while(listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

If we run this class, what will the output be?

Exercise 18.n)

Create a simple program that defines an ArrayList of integers, and that is filled with the first 50 even numbers in the most efficient way.

Exercise 18.o)

Which of the following statements are correct?

1. An implementation of Set cannot sort its elements.

2. An implementation of Set does not allow more than one null element.

3. The Set interface is extended by SortedSet.

4. The HashSet implementation is not thread safe.

Exercise 18.p)

Which of the following statements are correct?

1. The Vector class is an implementation of List.

2. An implementation of `List` does not allow null elements.

3. A `List` type reference can become thread safe if it is assigned a `List` type object that is returned by the `synchronizedList()` method.

4. To delete duplicate items from a list, just fill a `Set` with the elements of the `List`.

Exercise 18.q)

Given the class:

```java
import java.util.*;

public class ClaudioLinkedList extends LinkedList<String> {
    public ClaudioLinkedList() {
        add("X");
        add("L");
        add("W");
        add("U");
        add("D");
        add("I");
        add("Z");
    }
}
```

add in the following class `Exercise18Q`:

```java
import java.util.*;

public class Exercise18Q {

    public static void main(String args[]) {
        ClaudioLinkedList claudioLinkedList = new ClaudioLinkedList();
        /*INSERT CODE HERE*/
        System.out.println(claudioLinkedList);
    }
}
```

in place of the comment `/*INSERT CODE HERE*/`, the code which will allow you to generate the following output:

```
[C, L, A, U, D, I, O]
```

Exercise 18.r)

If we want to use a map in a thread-safe manner, what options do we have? List at least two options.

Exercise 18.s)

If we want to use an immutable list, what options do we have? List at least two options. And if we want to use an immutable set, what options do we have? List at least two options.

Exercise 18.t)

Given the list of strings that returns the following method:

```java
public static List<String> getStringList() {
    String string = "Les enfants qui s'aiment s'embrassent debout "
      + "Contre les portes de la nuit "
      + "Et les passants qui passent les désignent du doigt "
      + "Mais les enfants qui s'aiment "
      + "ne sont là pour personne "
      + "Et c'est seulement leur ombre "
      + "qui tremble dans la nuit "
      + "Excitant la rage des passants "
      + "Leur rage, leur mépris, leurs rires et leur envie "
      + "Les enfants qui s'aiment ne sont là pour personne "
      + "Ils sont ailleurs bien plus loin que la nuit "
      + "Bien plus haut que le jour "
      + "Dans l'éblouissante clarté de leur premier amour. ";
    String[] strings = string.split(" ");
    return Arrays.asList(strings);
}
```

write a pipeline that considers only words that do not begin with "a", and calculates (and prints) the average length.

Exercise 18.u)

Which of the following statements are correct?

1. `Optional` is a generic interface.
2. The method `ofNullable()` returns an `Optional` object that acts as a wrapper to the object passed as input.
3. `orElseThrow()` is an `Optional` method that returns an `Optional` object or throws an exception that can be specified in input, in case the "wrapped" object is `null`.
4. `findFirst()` is an `Optional` method that returns an `Optional` or `null` object, in case the "wrapped" object is `null`.

Exercise 18.v)

Given the following code:

```
import java.util.*;
import java.util.stream.*;

public class Exercise18V {

    public static void main(String args[]) {
        List<String> stringList = getStringList();
        /*INSERT CODE HERE*/
        System.out.println(map);
    }

    public static List<String> getStringList() {
        String string = "The children lovers embrace upright "
            + "Against night's doors "
            + "And passers-by who pass by point their finger at them "
            + "But the children lovers "
            + "Are there for no one "
            + "And it's only their shadow "
            + "Which quivers in the night "
            + "Stirring up the anger of the passers-by "
            + "Their anger, their contempt, their laughs and their desire "
            + "The children lovers are there for no one "
            + "They're elsewhere much further than the night "
            + "Much higher than the day "
            + "In the dazzling light of their first love. ";
        String[] strings = string.split(" ");
        return Arrays.asList(strings);
    }
}
```

Write a pipeline to create a map that groups the words of the text that begin with the same initial letter, ignoring the fact that the letter is uppercase or lowercase. Insert this pipeline instead of the comment /*INSERT CODE HERE*/.

Exercise 18.z)

Starting from the result of exercise 18.v, replace the printing instruction:

```
        System.out.println(map);
```

with a simple pipeline that prints the contents of the map line by line.

Exercise 18.aa)

Which of the following statements are correct?

1. The HashMap class extends Hashtable.

2. The Hashtable class is not thread safe.

3. The Hashtable class extends Properties.

4. Enumeration has been deprecated since enums and the enum keyword were introduced in Java 5.

5. The Enumeration interface has the same methods as the Iterator interface.

6. The Vector class implements List.

7. The Vector class is not thread safe.

8. The Vector class is not generic.

9. In general, the Vector class, being synchronized, offers superior performance compared to an ArrayList.

10. The Vector class is deprecated.

Chapter 18
Exercise Solutions

Collections Framework and Stream API

Solution 18.a) Framework Collections, True or False:

1. **True**.
2. **False**.
3. **True**.
4. **True**.
5. **True**.
6. **True**.
7. **True**.
8. **False**.
9. **True**.
10. **False**.

Solution 18.b) Stream API, True or False:

1. **False**, Stream is an interface.

2. **False**.

3. **True**.

4. **True**.

5. **False**, Optional is a class (also declared final and therefore not extensible).

6. **False**, are terminal operations.

7. **False**, the joining() method belongs to the Collectors class.

8. **False**.

9. **True**.

10. **False**.

Solution 18.c)

The custom exception could be the following:

```
package com.claudiodesio.excs;

public class DuplicateException extends RuntimeException {

    public DuplicateException(Object duplicateElement) {
        super("Unable to add item \""
                + duplicateElement + "\" because already present");
    }
}
```

While we could code the requested collection in this way:

```
package com.claudiodesio.collections;

import com.claudiodesio.excs.DuplicateException;
import java.util.HashSet;

public class RobustSet<E> extends HashSet<E> {

    @Override
    public boolean add(E e) {
        boolean result = super.add(e);
```

```
        if (!result) {
            throw new DuplicateException(e);
        }
        return result;
    }
}
```

And here's a test class:

```
package com.claudiodesio.test;

import com.claudiodesio.collections.RobustSet;
import com.claudiodesio.excs.DuplicateException;

public class RobustSetTest {

    public static void main(String args[]) {
        RobustSet<String> set = getRobustSet();
        try {
            set.add("Italy");
        } catch (DuplicateException duplicateException) {
            System.out.println(duplicateException.getMessage());
        }
        System.out.println(set);
    }

    public static RobustSet<String> getRobustSet() {
        RobustSet<String> set = new RobustSet<>();
        set.add("Italy");
        set.add("French");
        set.add ("Poland");
        set.add ("Germany");
        set.add ("England");
        set.add ("Spain");
        set.add ("Greece");
        set.add ("Netherlands");
        set.add ("Portugal");
        set.add ("Belgium");
        return set;
    }
}
```

The output will be:

```
Unable to add item "Italy" because already present
  [Germany, England, French, Belgium, Poland, Netherlands, Italy,
  Spain, Greece, Portugal]
```

Solution 18.d)

The code of the requested map could be the following:

```
package com.claudiodesio.collections;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;

public class IncrementalMap<K, V> extends HashMap<K, Collection<V>> {

    public void add(K key, V value) {
        if (this.get(key) == null) {
            Collection<V> collection = getCollection();
            collection.add(value);
            this.put(key, collection);
        } else {
            Collection<V> arrayList = this.get(key);
            arrayList.add(value);
        }
    }

    protected Collection<V> getCollection() {
        return new ArrayList<>();
    }
}
```

Note that the add() method is not an override (the method to add key-value pairs is the put() method) but an ad hoc method.
A test class follows:

```
package com.claudiodesio.test;

import com.claudiodesio.collections.IncrementalMap;
import com.claudiodesio.collections.RobustSet;
import java.util.Iterator;

public class IncrementalMapTest {

    public static void main(String args[]) {
        IncrementalMap<Integer, String> map = new IncrementalMap<>();
        filIncrementalMap( map);
        System.out.println(map);
    }

    public static void filIncrementalMap(IncrementalMap<Integer, String> map) {
        RobustSet<String> set = RobustSetTest.getRobustSet();
        int i = 1;
```

570

```
        int j = 1;
        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            if (i % 3 == 0) {
                j++;
            }
            String string = iterator.next();
            map.add(j, string);
            i++;
        }
    }
}
```

which generates the following output:

```
{1=[Greece, Netherlands], 2=[French, Belgium, Poland], 3=[England, Italy, Portugal],
 4=[Germany, Spain]}
```

Solution 18.e)

The code of the requested map could be the following:

```
package com.claudiodesio.collections;

import java.util.Collection;

public class RobustIncrementalMap<K, V> extends IncrementalMap<K, V> {

    @Override
    protected Collection<V> getCollection() {
        return new RobustSet<>();
    }
}
```

Note that this time the add() method is an override, and simply by substitute the ArrayList defined in the IncrementalMap class with a RobustSet we have solved the situation. If we want, we could do some refactoring on these two classes in order to improve our code. First, we return to the IncrementalMap class and rewrite the add() method with some small tricks:

```
package com.claudiodesio.collections;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;

public class IncrementalMap<K, V> extends HashMap<K, Collection<V>> {
```

```
    public void add(K key, V value) {
        if (this.get(key) == null) {
            Collection<V> collection = getCollection();
            collection.add(value);
            this.put(key, collection);
        } else {
            Collection<V> arrayList = this.get(key);
            arrayList.add(value);
        }
    }

    protected Collection<V> getCollection() {
        return new ArrayList<>();
    }
}
```

In this way we can simplify the subclass:

```
package com.claudiodesio.collections;

import java.util.Collection;

public class RobustIncrementalMap<K, V> extends IncrementalMap<K, V> {

    @Override
    protected Collection<V> getCollection() {
        return new RobustSet<>();
    }

}
```

And get the same result, without code duplication.
The test class follows:

```
package com.claudiodesio.test;

import com.claudiodesio.collections.IncrementalMap;
import com.claudiodesio.collections.RobustIncrementalMap;
import com.claudiodesio.excs.DuplicateException;

public class RobustIncrementalMapTest {

    public static void main(String args[]) {

        IncrementalMap<Integer, String> map = new RobustIncrementalMap<>();
        IncrementalMapTest.fillIncrementalMap(map);
        try {
            map.add(1, "Greece");
        } catch (DuplicateException duplicatoException) {
```

```
                System.out.println(duplicatoException.getMessage());
            }
            System.out.println(map);
        }
    }
```

Which generates the following output:

```
Unable to add item "Greece" because already present
  {1=[Greece, Netherlands], 2=[French, Belgium, Poland], 3=[England,
  Italy, Portugal], 4=[Germany, Spain]}
```

Solution 18.f)

In the City class we add the required methods:

```
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 19 * hash + Objects.hashCode(this.name);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final City other = (City) obj;
        if (!Objects.equals(this.name, other.name)) {
            return false;
        }
        return true;
    }
```

The StreamsTest class instead could be coded as follows:

```
package com.claudiodesio.test;

import com.claudiodesio.collections.RobustSet;
import com.claudiodesio.data.City;

public class StreamsTest {

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

573

```
        RobustSet<City> set = getRobustSet();
        System.out.println("City on the sea:");
        set.stream().filter(e->e.isOnTheSea()).forEach(System.out::println);
        System.out.println("\nState capital:");
        set.stream().filter(e->e.isStateCapital()).forEach(System.out::println);
        System.out.println("\nCities whose name ends with 'a':");
        set.stream().filter(
          e->e.getName().endsWith("a")).forEach(System.out::println);
    }

    public static RobustSet<City> getRobustSet() {
        RobustSet<City> set = new RobustSet<>();
        set.add(new City("Milano", true, false));
        set.add(new City("Rovigo", false, false));
        set.add(new City("Potenza", true, false));
        set.add(new City("Siracusa", false, true));
        set.add(new City("Perugia", true, false));
        set.add(new City("Napoli", true, true));
        set.add(new City("Pescara", false, true));
        set.add(new City("Taranto", false, true));
        set.add(new City("Siena", false, false));
        return set;
    }
}
```

The output follows l'output:

```
City on the sea:
Napoli
Siracusa
Taranto
Pescara

State capital:
Napoli
Potenza
Perugia
Milano

Cities whose name ends with 'a':
Potenza
Perugia
Siena
Siracusa
Pescara
```

Solution 18.g)

The correct statements are the numbers 2, 4 and 5.

574

Solution 18.h)

The correct answer is the number 3 because HashMap does not extend Collection, and therefore it is not possible to assign a Collection reference to a HashMap. The output of the compilation, is in fact the following:

```
Exercise18H.java:5: warning: [rawtypes] found raw type: Collection
        Collection map = new HashMap(10);
        ^
  missing type arguments for generic class Collection<E> where E is a type-variable:
    E extends Object declared in interface Collection
Exercise18H.java:5: warning: [rawtypes] found raw type: HashMap
        Collection map = new HashMap(10);
                             ^
  missing type arguments for generic class HashMap<K,V> where K,V are type-variables:
    K extends Object declared in class HashMap
    V extends Object declared in class HashMap
Exercise18H.java:5: error: incompatible types: HashMap cannot be converted to Collection
        Collection map = new HashMap(10);
                          ^
Exercise18H.java:6: error: cannot find symbol
        map.put(1,1);
           ^
  symbol:   method put(int,int)
  location: variable map of type Collection
2 errors
2 warnings
```

Note that two warnings are also shown because we used raw type, and that the errors are also two, because the reference map, being of the Collection type, does not declare the put() method (which is declared in the Map interface).

Solution 18.i)

The correct answers are the numbers 1 and 3. Iterator does not extend Iterable, so the statement 2 is incorrect. In the case of statement 4 the statement would have been correct if it had been: Collection **extends** the Iterable interface. In fact, both Collection and Iterable are interfaces and not classes. This implies that one can extend the other, not implement it.

Solution 18.l)

The only correct answer is the number 4. The statement number 1 is false simply because Collection is an interface and not a class. As for statement 2, the toArray() method is declared by the Collection interface. Furthermore, there is no toCollection() method in the Arrays class, so the statement 3 is also incorrect.

Solution 18.m)

The output of the `Exercise18M` class is as follows:

```
*
$
$
*
```

In fact, two elements are inserted (the strings `*` and `@`) with the method `add()` in the `list` list, and then with the method `set()` the string `@` is overwritten with the string `$`. The subsequent loops print the elements of the list by iterating with a `ListIterator` first forward and then backwards.

Solution 18.n)

The solution could be the following class:

```java
import java.util.*;

public class Exercise18N {
    public static void main(String args[]) {
        ArrayList<Integer> list = new ArrayList<>();
        list.ensureCapacity(50);
        long startTime = System.currentTimeMillis();
        for (int i = 1; i <=100000; ++i) {
            if (i%2==0) {
                list.add(i);
            }
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Time = " + (endTime - startTime));
    }
}
```

Solution 18.o)

All statements are correct except number 1. In fact, being `Set` extended by `SortedSet` (see statement number 3), the implementations of `SortedSet` as `TreeSet` will be ordered. The number 2 is true since any `Set` does not allow duplicates, so it is not possible to add the `null` element twice.

Solution 18.p)

Only the statements number 3 and 4 are correct.

576

Solution 18.q)

The solution could be the following class:

```java
import java.util.*;

public class Exercise18Q {

    public static void main(String args[]) {

        ClaudioLinkedList claudioLinkedList = new ClaudioLinkedList();
        /*INSERT CODE HERE*/
        claudioLinkedList.removeFirst();
        claudioLinkedList.addFirst("C");
        claudioLinkedList.set(2, "A");
        claudioLinkedList.removeLast();
        claudioLinkedList.addLast("O");
        System.out.println(claudioLinkedList);
    }
}
```

Solution 18.r)

We could use a `Hashtable`, a `ConcurrentHashMap`, or use a reference that points to the result of the invocation of a `synchronizedMap()` synchronizer method.

Solution 18.s)

We could use a reference that points to the result of invoking an `unmodifiableList()` synchronizer method, as shown below:

```java
List<String> immutableList = Arrays.asList("a", "b", "c");
immutableList = Collections.unmodifiableList(immutableList);
```

Or you can use the static convenience `of()` method of the `List` interface, introduced in Java 9:

```java
List immutableList = List.of("a", "b", "c");
```

The same goes for the immutable set implementations. Here are the two examples required:

```java
Set<String> immutableSet = new HashSet<>(Arrays.asList("a", "b", "c"));
immutableSet = Collections.unmodifiableSet(immutableSet);
```

and exploiting the `of ()` static method of the `Set` interface:

```java
Set<String> immutableSet = Set.of("a", "b", "c");
```

Solution 18.t)

A possible solution could be the following:

577

```
import java.util.*;
public class Exercise18T {
    public static void main(String args[]) {
        List<String> stringList = getStringList();
        Double average = stringList.stream().filter(
          s -> !s.startsWith("a")).mapToInt(
          String::length).average().getAsDouble();
        System.out.println(average);
     }
    public static List<String> getStringList() {
        String string = "Les enfants qui s'aiment s'embrassent debout "
          + "Contre les portes de la nuit "
          + "Et les passants qui passent les désignent du doigt "
          + "Mais les enfants qui s'aiment "
          + "ne sont là pour personne "
          + "Et c'est seulement leur ombre "
          + "qui tremble dans la nuit "
          + "Excitant la rage des passants "
          + "Leur rage, leur mépris, leurs rires et leur envie "
          + "Les enfants qui s'aiment ne sont là pour personne "
          + "Ils sont ailleurs bien plus loin que la nuit "
          + "Bien plus haut que le jour "
          + "Dans l'éblouissante clarté de leur premier amour. ";
        String[] strings = string.split(" ");
        return Arrays.asList(strings);
    }
}
```

Which will produce the following output:

```
4.607142857142857
```

Solution 18.u)

The correct statements are the numbers 2 and 3. The statement number 1 is incorrect because Optional is a generic class. The 4 is incorrect because findFirst() is declared in the Stream interface.

Solution 18.v)

The solution could be the following (the pipeline introduced is in bold):

```
import java.util.*;
import java.util.stream.*;
```

```
public class Exercise18V {

    public static void main(String args[]) {
        List<String> stringList = getStringList();
        Map<String, List<String>> map =
          stringList.stream().collect(Collectors.groupingBy(s ->
          (""+s.charAt(0)).toLowerCase()));
        System.out.println(map);
     }

    public static List<String> getStringList() {
        String string ="The children lovers embrace upright "
            + "Against night's doors "
            + "And passers-by who pass by point their finger at them "
            + "But the children lovers "
            + "Are there for no one "
            + "And it's only their shadow "
            + "Which quivers in the night "
            + "Stirring up the anger of the passers-by "
            + "Their anger, their contempt, their laughs and their desire "
            + "The children lovers are there for no one "
            + "They're elsewhere much further than the night "
            + "Much higher than the day "
            + "In the dazzling light of their first love. ";
        String[] strings = string.split(" ");
        return Arrays.asList(strings);
    }
}
```

Note that the Function passed to the groupingBy() method returns the first character, which is "added" to an empty string to be turned into a string, and then made lowercase to avoid case-sensitive distinctions. Unfortunately the output is not very readable (see next exercise):

```
{a=[Against, And, at, Are, And, anger, anger,, and, are], b=[by, But], c=[children,
 children, contempt,, children], d=[doors, desire, day, dazzling], e=[embrace,
 elsewhere], f=[finger, for, for, further, first], h=[higher], i=[it's, in, In], l=[lovers,
 lovers, laughs, lovers, light, love.], m=[much, Much], n=[night's, no, night, no, night],
 o=[one, only, of, one, of], p=[passers-by, pass, point, passers-by], q=[quivers],
 s=[shadow, Stirring], t=[The, their, them, the, there, their, the, the, the, Their,
 their, their, their, The, there, They're, than, the, than, the, the, their], u=[upright,
 up], w=[who, Which]}
```

Solution 18.z)

The solution could be the following (in bold the required pipeline):

```
import java.util.*;
import java.util.stream.*;

public class Exercise18Z {
```

```
    public static void main(String args[]) {
        List<String> stringList = getStringList();
        Map<String, List<String>> map =
          stringList.stream().collect(Collectors.groupingBy(s ->
          ("" + s.charAt(0)).toLowerCase()));
        map.entrySet().stream().forEach(System.out::println);
    }

    public static List<String> getStringList() {
        String string = "The children lovers embrace upright "
          + "Against night's doors "
          + "And passers-by who pass by point their finger at them "
          + "But the children lovers "
          + "Are there for no one "
          + "And it's only their shadow "
          + "Which quivers in the night "
          + "Stirring up the anger of the passers-by "
          + "Their anger, their contempt, their laughs and their desire "
          + "The children lovers are there for no one "
          + "They're elsewhere much further than the night "
          + "Much higher than the day "
          + "In the dazzling light of their first love. ";
        String[] strings = string.split(" ");
        return Arrays.asList(strings);
    }
}
```

The output is much clearer:

```
a=[Against, And, at, Are, And, anger, anger,, and, are]
b=[by, But]
c=[children, children, contempt,, children]
d=[doors, desire, day, dazzling]
e=[embrace, elsewhere]
f=[finger, for, for, further, first]
h=[higher]
i=[it's, in, In]
l=[lovers, lovers, laughs, lovers, light, love.]
m=[much, Much]
n=[night's, no, night, no, night]
o=[one, only, of, one, of]
p=[passers-by, pass, point, passers-by]
q=[quivers]
s=[shadow, Stirring]
t=[The, their, them, the, there, their, the, the, the, Their, their, their, their, The,
 there, They're, than, the, than, the, the, their]
u=[upright, up]
w=[who, Which]
```

Solution 18.aa)

None of the statements are correct!

580

Chapter 19
Exercises

Java Platform Module System

It will not be easy for programmers to start using modules. It is a concept that embraces a different IT branch: the *software architecture*. The exercises presented below aim to first clarify all the theoretical concepts, and then create modules in a progressive manner until a sustainable architecture is created, for the application that simulates an address book created with the exercises of the previous (and also the next) chapter.

Exercise 19.a)

Which of the following statements are correct:

1. With the modular system we can avoid runtime exceptions like `ClassNotFoundException`.

2. The strong encapsulation allows to make a certain package accessible only to the specified packages.

3. In JDK 9 the **rt.jar** file has been deleted.

4. The JVM Version 9, when running a program that uses modules, must manage them. This means that it will be less efficient than when it will run a program that does not use modules.

5. We can avoid using the modules because the concept of *anonymous modules* exists.

Exercise 19.b)

Which of the following statements are true?

1. A modular JAR file has a **.jmod** extension.

2. The JDK 9 has defined new applications as **jlink**, and many other JDK applications such as **jdeps** have been modified to support the introduction of the modules.

3. A module consists of a single file.

4. The classes of some packages like sun.*, and *.internal.* have been deleted.

5. An automatic module is a JAR file references int the module path.

Exercise 19.c)

Which of the following statements are true?

1. The words that define the module directives (module, open, opens, provides, requires, to, transitive, uses and with) are called "restricted words".

2. Both packages and modules follow the same convention for names.

3. It is possible to annotate the descriptor of a module **module-info.java** with some annotations.

4. The descriptor of a module **module-info.class** must be in the root directory of the module.

Exercise 19.d)

Explain what it means to execute the following command:

```
javac -d mods/com.domain.mymodule src/com.domain.mymodule/com/domain/*
    src/com.domain.mymodule/module-info.java
```

Exercise 19.e)

Explain what it means to execute the following command:

```
java --module-path mods -m com.domain/com.domain.HelloModularWorld
```

582

Exercise 19.f)

Define the following concepts regarding the software architecture:

1. **Software component**
2. **Vertical partitioning**
3. **Framework (horizontal partitioning)**
4. **Cohesion**
5. **Coupling**
6. **Subsystem**
7. **Dependency inversion principle**

Exercise 19.g)

Imagine that we have created a management software for a computer store. Let's also imagine having a module that contains the code that manages the billing of the store, and let's call it `shop.billing`. This module contains the packages `shop.billing.items`, `shop.billing.internalalgorithms` and `shop.billing.availablefunctions`. Then consider another module that contains the code that represents the graphic user interface of the application, and let's call it a `shop.gui`. Declare the relative descriptors that explain the dependency that exists between these two modules.

Exercise 19.h)

Starting from the solution of the previous exercise, declare the descriptor of a module that contains the code that allows to sell items called the `shop.sales`, and possibly modify the descriptors already defined. Keep in mind that:

1. The `shop.sales` module contains the packages called:
   - `shop.sales.availablefunctions`
   - `shop.sales.items`
   - `shop.sales.internalalgorithms`
2. From the graphical user interface it will be possible to sell items of the store.
3. At the same time as the sale, it must be possible to invoice the sold items.

Exercise 19.i)

Starting from the solution of the previous exercise, what can we do if we want the shop.billing module read the shop.gui.

Exercise 19.l)

Starting from the solution of the Exercise 19.h, what we can do to allow the shop.gui module to access through reflection to a private method defined in the package shop.billing.internalalgorithms?

Exercise 19.m)

Consider the solution of the previous exercise. If it is not possible to modify our code anymore, but we have noticed that a class of the package shop.sales must use a private method of the package shop.billing.internalalgorithms through reflection, what can we do?

Exercise 19.n)

Which of the following statements is correct:

1. The ServiceLoader class was introduced already in Java Version 6.

2. The Service Provider Interface (SPI) component depends on its implementations.

3. With the ServiceLoader class we can completely eliminate the dependency between modules.

4. To implement a service with ServiceLoader, implementations of a service provider interface must be exported from the respective modules.

5. A provider method is a kind of factory method.

Exercise 19.o)

Starting from the solution of Exercise 20.n, define packages for the various classes. Then create and compile a module that exports the package that contains all the classes that represent data.

Exercise 19.p)

Starting from the solution of Exercise 19.o, create a module that exports the package that contains the exceptions.

To compile multiple modules with a single command, the `--module-source-path` attribute is usually used. For example, to compile the exercise of section 19.3 on the `ServiceLoader`, we used the following command:

```
javac -d mods --module-source-path
  src src/com.claudiodesio.spi/module-info.java
  src/com.claudiodesio.spi/com/claudiodesio/spi/*
  src/com.claudiodesio.invs/module-info.java
  src/com.claudiodesio.invs/com/claudiodesio/invs/*
  src/com.claudiodesio.certs/module-info.java
  src/com.claudiodesio.certs/com/claudiodesio/certs/*
  src/com.claudiodesio.factory/module-info.java
  src/com.claudiodesio.factory/com/claudiodesio/factory/*
  src/com.claudiodesio.handlers/module-info.java
  src/com.claudiodesio.handlers/com/claudiodesio/handlers/*
  src/com.claudiodesio.client/module-info.java
  src/com.claudiodesio.client/com/claudiodesio/client/*
```

(written all on one line without wrapping).

Exercise 19.q)

Starting from the solution of Exercise 19.p, create a module that exports the package containing the utility classes.

Exercise 19.r)

Starting from the solution of Exercise 19.q, create a module that exports the package containing the class that plays the role of service provider interface. Our goal is to transform the way we serialize contacts with `ServiceLoader`.

Exercise 19.s)

Below is the `Exercise20N` class which represents the client of the phone book that we will create in the exercises of Chapter 20:

> **You can safely ignore the implementation details of the Input-Output APIs that are used below. You just have to know that we will use two different APIs (named IO and NIO2) to save objects on our hard disk. Since you'll have to understand and solve these exercises in the next chapter, you can come back later and give a look at the code you will write in the next exercises. Please focus on the argument of Chapter 19: modules.**

585

```
public class Exercise20N {

    private SerializationManager<Contact> fileManager;

    private Contact[] contacts;

    Exercise20N() {
        contacts = getContacts();
        fileManager = new FileNIO2Manager();
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS");
        retrieveContacts();
        System.out.println(
          "TESTING THE CREATION OF A CONTACT ALREADY EXISTING");
        createExistingContact();
        System.out.println("TRYING TO RETRIEVE A CONTACT");
        retrieveNonExistentContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateExistingContact();
        System.out.println("REMOVING AN EXISTING CONTACT");
        removeExistingContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateNonExistentContact();
        System.out.println("REMOVING A NON-EXISTENT CONTACT");
        removeNonExistentContact();
    }

    public void createExistingContact() {
        execute(()->fileManager.insert(contacts[0]));
    }

    public void updateExistingContact() {
        execute(()->fileManager.update(new Contact("Daniele",
            "Mics Street 1","07890")));
    }

    public void removeExistingContact() {
        execute(()->fileManager.remove(contacts[2].getName()));
    }

    public void updateNonExistentContact() {
        execute(()->fileManager.update(new Contact("Foo",
            "Mics Street 1","07890")));
    }
```

```java
    public void removeNonExistentContact() {
        execute(()->fileManager.remove("Ligeia"));
    }

    public void retrieveNonExistentContact() {
        execute(()->fileManager.retrieve("Foo"));
    }

    public void createContacts() {
        for (Contact contact : contacts) {
            System.out.println("Creating contact:\n" + contact);
            createContact(contact);
        }
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            System.out.println("Retrieving contact: " + contact.getName());
            retrieveContact(contact.getName());
        }
    }

    public void retrieveContact(String nameContact) {
        execute(()->fileManager.retrieve(nameContact));
    }

    private void createContact(Contact contact) {
        execute(()->fileManager.insert(contact));
    }

    private Contact[] getContacts() {
        Contact contact1 = new Contact("Daniele","Guitars Street 1","01234560");
        Contact contact2 = new Contact("Giovanni","Sciences Avenue 2","0565432190");
        Contact contact3 = new Contact("Ligeia","Secrets Place 3","07899921");
        Contact[] contacts = {
            contact1, contact2, contact3
        };
        return contacts;
    }

    public <O> O execute(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.execute();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }
```

```
    public void execute(Executor executor) {
        try {
            executor.execute();
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public static void main(String args[]) {
        Exercise20N exercise20N = new Exercise20N();
        exercise20N.executeTest();
    }
}
```

We have printed in bold instructions on which to focus our attention. Notice how in the constructor an object of the class `FileManagerNIO2` is explicitly created, and assigned to a reference of `SerializationManager`. Our goal for this exercise, is to create two different modules to transform into services the two ways in which we serialize the `Contact` objects as we see in chapter 20, with the `FileManager` and `NIO2Manager` classes. In the next exercises we will use these modules.

Exercise 19.t)

Starting from the solution of Exercise 19.s, rename the Exercise20N class to Exercise19T, and use the `ServiceLoader` to load the services. Make it possible to specify from the command line which serialization manager should be used. Then create a module that contains this class and run the application.

Exercise 19.u)

Starting from the solution of Exercise 19.t, create a factory class and the relative module and move the `getSerializationManager()` method present in the Exercise19T class inside it. Modify the `com.claudiodesio.phonebook.test` module descriptor accordingly.

Exercise 19.v)

Pack the modules created in the corresponding modular JARs and run the application.

Exercise 19.z)

With **jlink** create a custom environment with only the `java.base` module. Then copy in a **lib** folder the solutions of the Exercise 19.v, run our executable modular JAR via the newly created runtime.

588

Chapter 19
Exercise Solutions

Java Platform Module System

The correct statements are the numbers 1, 3 and 5. The statement number 2 is incorrect because the strong encapsulation allows to make a certain package accessible only to the specified modules (not to the specified packages). The statement number 4 is incorrect because as stated in section 19.1.1, the JVM optimization techniques are more effective if the types that will be used in the application are known a priori.

Solution 19.b)

The only correct statements are the numbers 2 and 5. The number 1 is incorrect because a modular JAR has a **.jar** extension. A file with the **.jmod** extension is instead characterized by the fact that it also contains native resources. The statement number 3 is clearly false: the descriptor of the module is a single file, not the module itself. Finally, the statement number 4 is also false because the packages were not eliminated, but made unavailable using strong encapsulation.

Solution 19.c)

All statements are correct.

Solution 19.d)

With the following instruction:

```
javac -d mods/com.domain.mymodule
  src/com.domain.mymodule/com/domain/*
  src/com.domain.mymodule/module-info.java
```

we are compiling a module named `com.domain.mymodule`.
In particular, with the option:

```
-d mods/com.domain.mymodule
```

we are requesting to the **javac** command, that the result of the compilation will have to be placed inside the subfolder **com.domain.mymodule** of the **mods** directory (folder that must be present in the same position where we are executing the compilation command).
With the argument:

```
src/com.domain.mymodule/com/domain/*
```

we are specifying that all the files present in the **src/com.domain.mymodule/com/domain** path must be compiled.
With:

```
src/com.domain.mymodule/module-info.java
```

we are specifying that the descriptor of the module **module-info.java** present in the **src/com.domain.mymodule** directory must also be compiled.

Solution 19.e)

With the following instruction:

```
java --module-path mods -m com.domain/com.domain.mymodule.HelloModularWorld
```

we are running the `main()` method of the `com.domain.mymodule.HelloModularWorld` class of the module named `com.domain.mymodule`, specifying the **mods** directory as module path.

Solution 19.f)

A **software component** is a set of classes with a well-defined interface, which provides executable functions regardless of the context. A software component is therefore an executable and reusable subsystem.

590

In a **vertical partitioning** the application is divided by features of the same importance, and each of them can be developed independently of the other.

Horizontal partitioning is a partitioning based primarily on inheritance. A typical example is a framework, that is a micro-architecture that provides an extensible model for implementing applications within a specific domain.

Cohesion is the measure of how much a certain element (class, method, package, subsystem, etc.) contributes to achieving a certain purpose within the system.

Coupling is the metric that measures the dependence between classes, between packages, between methods and so on.

A **subsystem** is a set of classes linked by associations, events and constraints, and for which an independent development of the other subsystems is possible. Subsystems are internally highly cohesive (i.e. they declare only types that work together to achieve a certain purpose), internally there are highly coupled types that are loosely coupled with external types.

The **Dependency Inversion Principle** states that classes must depend on abstractions and not implementations.

Solution 19.g)

A solution could be the following:

```
module shop.gui {
    requires shop.billing;
}
```

and:

```
module shop.billing {
    exports shop.billing.items;
    exports shop.billing.availablefunctions;
    //exports shop.billing.internalalgorithms;
}
```

That is, the `shop.gui` module reads the `shop.billing` module, which in turn exposes only two of the three packages (note that the third directive is commented out). In fact, the `shop.billing.internalalgorithms` package, with that name, very probably could not be used directly from the outside.

Solution 19.h)

Moving in such an abstract context, it is possible to hypothesize different solutions. Assuming that the descriptor of the `shop.billing` module will not be modified, we could simply export the right packages from the `shop.sales` module:

```
module shop.sales {
    exports shop.sales.items;
    exports shop.sales.availablefunctions;
    //exports shop.sales.internalalgorithms;
}
```

and make the shop.gui module read also shop.sales:

```
module shop.gui {
    requires shop.billing;
    requires shop.sales;
}
```

thus delegating to the graphical user interface, the burden of defining a function that contextualises the sale and invoicing of an item in a single request from the user. Since it is not a good practice to assign business rules to a graphical user interface (which must already implement the presentation logic), we decide to make the situation more flexible, so that we can make decisions later. Then using the `requires transitive` directive, we are going to modify the descriptor of the shop.sales module, so that it can read (and make read) the shop.billing module.

```
module shop.sales {
    exports shop.sales.items;
    exports shop.sales.availablefunctions;
    //exports shop.sales.internalalgorithms;
    requires transitive shop.billing;
}
```

At this point the shop.gui module can only read shop.sales, as it will transitively read also shop.billing:

```
module shop.gui {
    //requires shop.billing;
    requires shop.sales;
}
```

Solution 19.i)

Actually, nothing can be done unless you want to re-evaluate all the dependencies already specified. If we wanted shop.billing to read shop.gui interface, we would get a cyclic dependency error.

Solution 19.l)

The solution is to review the descriptor of the shop.billing module by exporting the shop.sales.internalalgorithms package, and at the same time open it to the shop.gui module.

592

```
module shop.billing {
    exports shop.billing.items;
    exports shop.billing.availablefunctions;
    exports shop.billing.internalalgorithms;
    opens shop.billing.internalalgorithms to shop.gui;
}
```

Solution 19.m)

The only solution is to run the program by specifying from the command line the opening of the shop.billing module to the shop.sales module, with the following syntax:

```
-add-opens shop.billing/shop.billing.internalalgorithms=shop.sales
```

Solution 19.n)

The correct statements are the numbers 1, and 5. The statement number 2 is false, because the implementations depend on the service provider interface. The statement number 4 is false as it is not necessary to export the implementations of the service provider interface, but rather to use the provides to directive.

Solution 19.o)

A possible solution is to create a module that we will call com.claudiodesio.phonebook.data. It will contain the Contact and Data classes, both belonging to a package named as the module:

```
package com.claudiodesio.phonebook.data;

import java.io.Serializable;

public interface Data extends Serializable {
}
```

and:

```
package com.claudiodesio.phonebook.data;

import java.io.Serializable;

public class Contact implements Data {

    private static final long serialVersionUID = 8942402240056525661L;

    private String name;
```

593

```
    private String address;

    private String phoneNumber;

    public Contact (String name, String address, String phoneNumber) {
        this.name = name;
        this.address = address;
        this.phoneNumber = phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString(){
        return "Name:\t" + name + "\nAddress:\t" + address + "\nPhone:\t"
          + phoneNumber;
    }
}
```

We can see the packages of the other classes in the solutions of the next exercises.
The descriptor of the module will be the following:

```
module com.claudiodesio.phonebook.data {
    exports com.claudiodesio.phonebook.data;
}
```

To compile the module we will use the following command with the usual folder structure that

we have used so far (the complete exercise is in the **Code\chapter_19\exercises\19.o** folder containing the source code of the book, that you have probably downloaded together with the file that you are reading, at **http://www.javaforaliens.com**):

```
javac -d mods --module-source-path src
 src/com.claudiodesio.phonebook.data/module-info.java
 src/com.claudiodesio.phonebook.data/com/claudiodesio/phonebook/data/*.java
```

Solution 19.p)

A possible solution is to create a module that we will name:

    com.claudiodesio.phonebook.exceptions

It will contain the NonExistentContactException and DuplicateContactException classes, both belonging to a package named as the module:

```
package com.claudiodesio.phonebook.exceptions;

import java.io.IOException;

public class NonExistentContactException extends IOException {

    private static final long serialVersionUID = 8942402240056525663L;

    public NonExistentContactException(String message) {
        super(message);
    }
}
```

and:

```
package com.claudiodesio.phonebook.exceptions;

import java.io.IOException;

public class DuplicateContactException extends IOException {

    private static final long serialVersionUID = 8942402240056525662L;

    public DuplicateContactException (String message) {
        super(message);
    }
}
```

The module descriptor will be the following:

```
module com.claudiodesio.phonebook.exceptions {
    exports com.claudiodesio.phonebook.exceptions;
}
```

595

To compile both the modules, we will use the following command with the usual folder structure that we have used so far (the complete exercise is in the **Code\chapter_19\exercises\19.p** folder of the zip file containing the source code of the book):

```
javac -d mods --module-source-path src
  src/com.claudiodesio.phonebook.data/module-info.java
  src/com.claudiodesio.phonebook.data/com/claudiodesio/phonebook/data/*.java
  src/com.claudiodesio.phonebook.exceptions/module-info.java
  src/com.claudiodesio.phonebook.exceptions/com/claudiodesio/phonebook/exceptions/*.java
```

Solution 19.q)

A possible solution is to create a module that we will name `com.claudiodesio.phonebook.util`. It will contain the `Retriever`, `Executor` and `FileUtils` classes, belonging to a package named as the module:

```
package com.claudiodesio.phonebook.util;

@FunctionalInterface
public interface Retriever<O> {

    O execute() throws Exception;
}
```

and:

```
package com.claudiodesio.phonebook.util;

@FunctionalInterface
public interface Executor {

    void execute() throws Exception;
}
```

and:

```
package com.claudiodesio.phonebook.util;

public class FileUtils {
    public static final String SUFFIX = ".con";

    public static String getFileName(String name) {
        return name + SUFFIX;
    }
}
```

The module descriptor will be the following:

```
module com.claudiodesio.phonebook.util {
    exports com.claudiodesio.phonebook.util;
}
```

596

To compile the three modules, we will use the following command with the usual folder structure that we have used so far (the complete exercise is in the **Code\chapter_19\exercises\19.q** folder of the file containing the source code of the book):

```
javac -d mods --module-source-path src
 src/com.claudiodesio.phonebook.data/module-info.java
 src/com.claudiodesio.phonebook.data/com/claudiodesio/phonebook/data/*.java
 src/com.claudiodesio.phonebook.exceptions/module-info.java
 src/com.claudiodesio.phonebook.exceptions/com/claudiodesio/phonebook/exceptions/*.java
 src/com.claudiodesio.phonebook.util/module-info.java
 src/com.claudiodesio.phonebook.util/com/claudiodesio/phonebook/util/*.java
```

Solution 19.r)

`com.claudiodesio.phonebook.spi.SerializationManager` is the class will act as service provider interface, and which will belong to the `com.claudiodesio.phonebook.spi` module:

```
package com.claudiodesio.phonebook.spi;

import com.claudiodesio.phonebook.data.Data;

import java.io.*;

import java.util.*;

public interface SerializationManager<T extends Data> {

    void insert(T data) throws IOException;

    T retrieve(String id) throws IOException, ClassNotFoundException;

    void update(T data) throws IOException;

    void remove(String id) throws IOException;
}
```

The descriptor of the module will be the following:

```
module com.claudiodesio.phonebook.spi {
    exports com.claudiodesio.phonebook.spi;
    requires com.claudiodesio.phonebook.data;
}
```

In this case it was necessary to read the `com.claudiodesio.phonebook.data` module, since the `SerialializationManager` class uses the `Data` interface.
To compile the four modules, we will use the following command with the usual folder struc-

ture that we have used so far (the complete exercise is in the **Code\chapter_19\exercises\19.r** folder of the file containing the source code of the book):

```
javac -d mods --module-source-path src
 src/com.claudiodesio.phonebook.data/module-info.java
 src/com.claudiodesio.phonebook.data/com/claudiodesio/phonebook/data/*.java
 src/com.claudiodesio.phonebook.exceptions/module-info.java
 src/com.claudiodesio.phonebook.exceptions/com/claudiodesio/phonebook/exceptions/*.java
 src/com.claudiodesio.phonebook.util/module-info.java
 src/com.claudiodesio.phonebook.util/com/claudiodesio/phonebook/util/*.java
 src/com.claudiodesio.phonebook.spi/module-info.java
 src/com.claudiodesio.phonebook.spi/com/claudiodesio/phonebook/spi/*.java
```

Solution 19.s)

First of all, we had to add to the `FileManager` and `FileManagerNIO2` classes, in addition to the package declarations, several `import` declarations:

```java
package com.claudiodesio.phonebook.io;

import com.claudiodesio.phonebook.spi.SerializationManager;
import com.claudiodesio.phonebook.data.Contact;
import com.claudiodesio.phonebook.exceptions.*;
import com.claudiodesio.phonebook.util.*;
import java.util.*;
import java.io.*;

public class FileManager implements SerializationManager<Contact> {

    @Override
    public void insert(Contact contact) throws DuplicateContactException,
      FileNotFoundException, IOException {
        Contact duplicateContact = getContact(contact.getName());
        if (duplicateContact != null) {
            throw new DuplicateContactException(contact.getName() +
              ": contact already exists!");
        }
        store(contact);
    }

    @Override
    public Contact retrieve(String name) throws ContactNotFoundException,
      DuplicateContactException {
        Contact contact = getContact(name);
        if (contact == null) {
            throw new ContactNotFoundException(name + ": contact not found!");
        }
        return contact;
    }
```

```java
        @Override
        public void update(Contact contact) throws ContactNotFoundException,
                DuplicateContactException, FileNotFoundException, IOException {
            if (isContactExisting(contact.getName())) {
                store(contact);
            } else {
                throw new ContactNotFoundException(contact.getName() +
                  ": contact not found!");
            }
        }

        @Override
        public void remove(String name) throws ContactNotFoundException,
          DuplicateContactException, FileNotFoundException, IOException {
            File file = new File(FileUtils.getFileName(name));
            if (file.delete()) {
                System.out.println("Contact " + name + " deleted!");
            } else {
                throw new ContactNotFoundException(name + ": contact not found!");
            }
        }

        private void store(Contact contact) throws  FileNotFoundException,
          IOException {
            try (FileOutputStream fos =
            new FileOutputStream (new File(FileUtils.getFileName(
              contact.getName()))));
            ObjectOutputStream s = new ObjectOutputStream (fos);) {
                s.writeObject (contact);
                System.out.println("Contact stored:\n"+ contact);
            }
        }

        private boolean isContactExisting(String name) {
            File file = new File(FileUtils.getFileName(name));
            return file.exists();
        }

        private Contact getContact(String name) {
            try (FileInputStream fis = new FileInputStream (
              new File(FileUtils.getFileName(name)));
            ObjectInputStream ois = new ObjectInputStream (fis);) {
                Contact contact = (Contact)ois.readObject();
                System.out.println("Contact retrieved:\n"+ contact);
                return contact;
            } catch (Exception exc) {
                return null;
            }
        }
    }
}
```

Java for Aliens - Claudio De Sio Cesari (www.javaforaliens.com)

and:

```
package com.claudiodesio.phonebook.nio;

import com.claudiodesio.phonebook.spi.SerializationManager;
import com.claudiodesio.phonebook.data.Contact;
import com.claudiodesio.phonebook.exceptions.*;
import com.claudiodesio.phonebook.util.*;
import java.util.*;
import java.io.*;
import java.util.*;
import java.io.*;
import java.nio.file.*;

public class FileManagerNIO2 implements SerializationManager<Contact> {

    @Override
    public void insert(Contact contact) throws DuplicateContactException,
      FileNotFoundException, IOException {

        Path path = Paths.get(FileUtils.getFileName(contact.getName()));
        if (Files.exists(path)) {
            throw new DuplicateContactException(contact.getName() +
              ": contact already exists!");
        }
        store(contact);
    }

    @Override
    public Contact retrieve(String name) throws ContactNotFoundException,
      DuplicateContactException {

        Contact contact = getContact(name);
        if (contact == null) {
            throw new ContactNotFoundException(name + ": contact not found!");
        }
        return contact;
    }

    @Override
    public void update(Contact contact) throws ContactNotFoundException,
      DuplicateContactException, FileNotFoundException, IOException {

        if (isExistingContact(contact.getName())) {
            store(contact);
        } else {
            throw new ContactNotFoundException(contact.getName() +
              ": contact not found!");
        }
    }
```

600

```java
    @Override
    public void remove(String name) throws ContactNotFoundException,
      DuplicateContactException, FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getFileName(name));
        if (Files.exists(path)) {
            Files.delete(path);
            System.out.println("Contact "+ name +" deleted!");
        } else {
            throw new ContactNotFoundException(name +": contact not found!");
        }
    }

    private void store(Contact contact) throws  FileNotFoundException,
      IOException {
        Path path = Paths.get(FileUtils.getFileName(contact.getName()));
        Files.write(path, getBytesFromObject(contact));
        System.out.println("Contact stored:\n"+ contact);
    }

    private byte[] getBytesFromObject(Object object) throws IOException {
        try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
             ObjectOutput out = new ObjectOutputStream(bos)) {
            out.writeObject(object);
            return bos.toByteArray();
        }
    }

    private Object getObjectFromByte(byte[] bytes) throws IOException,
      ClassNotFoundException {
        try (ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
             ObjectInput in = new ObjectInputStream(bis)) {
            return in.readObject();
        }
    }

    private boolean isExistingContact(String name) {
        Path path = Paths.get(FileUtils.getFileName(name));
        return Files.exists(path);
    }

    private Contact getContact(String name) {
        Path path = Paths.get(FileUtils.getFileName(name));
        byte[] bytes = null;
        Contact contact = null;
        try {
            bytes = Files.readAllBytes(path);
            contact = (Contact)getObjectFromByte(bytes);
            System.out.println("Contact retrieved:\n" + contact);
        }
```

Java for Aliens - Claudio De Sio Cesari (www.javaforaliens.com)

```
        catch (Exception exc) {
            return null;
        }
        return contact;
    }
}
```

The first module will contain `FileManager` and will be called `com.claudiodesio.phonebook.io`, while `FileManagerNIO2` will be contained in the module `com.claudiodesio.phonebook.nio`. For the first module we can create the following descriptor:

```
module com.claudiodesio.phonebook.io {
    //exports com.claudiodesio.phonebook.io;
    provides com.claudiodesio.phonebook.spi.SerializationManager with
      com.claudiodesio.phonebook.io.FileManager;
    requires com.claudiodesio.phonebook.data;
    requires com.claudiodesio.phonebook.exceptions;
    requires com.claudiodesio.phonebook.spi;
    requires com.claudiodesio.phonebook.util;
}
```

While for the second we can create this other:

```
module com.claudiodesio.phonebook.nio {
    //exports com.claudiodesio.phonebook.nio;
    provides com.claudiodesio.phonebook.spi.SerializationManager with
      com.claudiodesio.phonebook.nio.FileManagerNIO2;
    requires com.claudiodesio.phonebook.data;
    requires com.claudiodesio.phonebook.exceptions;
    requires com.claudiodesio.phonebook.spi;
    requires com.claudiodesio.phonebook.util;
}
```

To compile the six modules, we will use the following command with the usual folder structure that we have used so far (the complete exercise is in the **Code\chapter_19\exercises\19.s** folder of the file containing the source code of the book):

```
javac -d mods --module-source-path src
 src/com.claudiodesio.phonebook.data/module-info.java
 src/com.claudiodesio.phonebook.data/com/claudiodesio/phonebook/data/*.java
 src/com.claudiodesio.phonebook.exceptions/module-info.java
 src/com.claudiodesio.phonebook.exceptions/com/claudiodesio/phonebook/exceptions/*.java
 src/com.claudiodesio.phonebook.util/module-info.java
 src/com.claudiodesio.phonebook.util/com/claudiodesio/phonebook/util/*.java
 src/com.claudiodesio.phonebook.spi/module-info.java
 src/com.claudiodesio.phonebook.spi/com/claudiodesio/phonebook/spi/*.java
 src/com.claudiodesio.phonebook.io/module-info.java
 src/com.claudiodesio.phonebook.io/com/claudiodesio/phonebook/io/*.java
 src/com.claudiodesio.phonebook.nio/module-info.java
 src/com.claudiodesio.phonebook.nio/com/claudiodesio/phonebook/nio/*.java
```

We have changed the name of the `Exercise18N` class with `Exercise19T` to exploit the `ServiceLoader` and load the services declared in the previous exercise. In addition, we have refactored the various imports and declared the belonging package.

```java
package com.claudiodesio.phonebook.test;

import java.util.function.*;
import com.claudiodesio.phonebook.spi.SerializationManager;
import com.claudiodesio.phonebook.data.Contact;
import com.claudiodesio.phonebook.util.*;
import java.util.Iterator;
import java.util.ServiceLoader;

public class Exercise19T {

    private SerializationManager<Contact> fileManager;

    private Contact[] contacts;

    public Exercise19T(String className) {
        contacts = getContacts();
        fileManager = getSerializationManager(className);
    }

    public SerializationManager<Contact> getSerializationManager(
      String className) {
        ServiceLoader<SerializationManager> serviceLoader = ServiceLoader.load(
          com.claudiodesio.phonebook.spi.SerializationManager.class);
        for (SerializationManager<Contact> serializationManager : serviceLoader)
        {
            if (serializationManager.getClass().
              getSimpleName().equals(className)) {
                return serializationManager;
            }
        }
        throw new IllegalArgumentException(
          "No serialization manager found for class = " + className);
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS");
        retrieveContacts();
        System.out.println(
          "TESTIING THE CREATION OF A CONTACT ALREADY EXISTING");
```

```
        createExistingContact();
        System.out.println("TRYING TO RETRIEVE A CONTACT");
        retrieveNonExistentContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateExistingContact();
        System.out.println("REMOVING AN EXISTING CONTACT");
        removeExistingContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateNonExistentContact();
        System.out.println("REMOVING A NON-EXISTENT CONTACT");
        removeNonExistentContact();
    }

    public void createExistingContact() {
        execute(()->fileManager.insert(contacts[0]));
    }

    public void updateExistingContact() {
        execute(()->fileManager.update(new Contact("Daniele",
          "Mics Street 1","07890")));
    }

    public void removeExistingContact() {
        execute(()->fileManager.remove(contacts[2].getName()));
    }

    public void updateNonExistentContact() {
        execute(()->fileManager.update(new Contact("Foo",
          "Mics Street 1","07890")));
    }

    public void removeNonExistentContact() {
        execute(()->fileManager.remove("Ligeia"));
    }

    public void retrieveNonExistentContact() {
        execute(()->fileManager.retrieve("Foo"));
    }

    public void createContacts() {
        for (Contact contact : contacts) {
            System.out.println("Creating contact:\n" + contact);
            createContact(contact);
        }
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            System.out.println("Retrieving contact: " + contact.getName());
```

```
                retrieveContact(contact.getName());
        }
    }

    public void retrieveContact(String nameContact) {
        execute(()->fileManager.retrieve(nameContact));
    }

    private void createContact(Contact contact) {
        execute(()->fileManager.insert(contact));
    }

    private Contact[] getContacts() {
        Contact contact1 = new Contact("Daniele","Guitars Street 1","01234560");
        Contact contact2 = new Contact("Giovanni", "Sciences Avenue 2","0565432190");
        Contact contact3 = new Contact("Ligeia","Secrets Place 3","07899921");
        Contact[] contacts = {
            contact1, contact2, contact3
        };
        return contacts;
    }

    public <O> O execute(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.execute();
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }

    public void execute(Executor executor) {
        try {
            executor.execute();
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public static void main(String args[]) {
        Exercise19T exercise19T = new Exercise19T(args[0]);
        exercise19T.executeTest();
    }
}
```

In bold we have highlighted the getSerializationManager() factory method which

605

manages which service to use (based on the `className` parameter). Then we defined the relative module of which we report the descriptor below, which can access via reflection to `SerializationManager`:

```
module com.claudiodesio.phonebook.test {
    uses com.claudiodesio.phonebook.spi.SerializationManager;
    requires com.claudiodesio.phonebook.spi;
    requires com.claudiodesio.phonebook.data;
    requires com.claudiodesio.phonebook.util;
}
```

We can now compile the seven modules defined so far with the following command (as always, the complete exercise is in the **Code\chapter_19\exercises\19.t** folder of the zip file containing the source code of the book).

```
javac -d mods --module-source-path src
 src/com.claudiodesio.phonebook.data/module-info.java
 src/com.claudiodesio.phonebook.data/com/claudiodesio/phonebook/data/*.java
 src/com.claudiodesio.phonebook.exceptions/module-info.java
 src/com.claudiodesio.phonebook.exceptions/com/claudiodesio/phonebook/exceptions/*.java
 src/com.claudiodesio.phonebook.util/module-info.java
 src/com.claudiodesio.phonebook.util/com/claudiodesio/phonebook/util/*.java
 src/com.claudiodesio.phonebook.spi/module-info.java
 src/com.claudiodesio.phonebook.spi/com/claudiodesio/phonebook/spi/*.java
 src/com.claudiodesio.phonebook.io/module-info.java
 src/com.claudiodesio.phonebook.io/com/claudiodesio/phonebook/io/*.java
 src/com.claudiodesio.phonebook.nio/module-info.java
 src/com.claudiodesio.phonebook.nio/com/claudiodesio/phonebook/nio/*.java
 src/com.claudiodesio.phonebook.test/module-info.java
 src/com.claudiodesio.phonebook.test/com/claudiodesio/phonebook/test/*.java
```

To run the application we can use the following command:

```
java --module-path mods -m
 com.claudiodesio.phonebook.test/com.claudiodesio.phonebook.test.Exercise19T FileManager
```

Which will print the same output seen in the solution of the 18.n exercise and which we report below for convenience:

```
TESTING THE CREATION OF THE THREE CONTACTS
Creating contact:
Name:   Daniele
Address:        Guitars Street 1
Phone:  01234560
Contact stored:
Name:   Daniele
Address:        Guitars Street 1
Phone:  01234560
Creating contact:
Name:   Giovanni
```

```
Address:        Sciences Avenue 2
Phone:  0565432190
Contact stored:
Name:   Giovanni
Address:        Sciences Avenue 2
Phone:  0565432190
Creating contact:
Name:   Ligeia
Address:        Secrets Place 3
Phone:  07899921
Contact stored:
Name:   Ligeia
Address:        Secrets Place 3
Phone:  07899921
RETRIEVING THE THREE CONTACTS
Retrieving contact: Daniele
Contact retrieved:
Name:   Daniele
Address:        Guitars Street 1
Phone:  01234560
Retrieving contact: Giovanni
Contact retrieved:
Name:   Giovanni
Address:        Sciences Avenue 2
Phone:  0565432190
Retrieving contact: Ligeia
Contact retrieved:
Name:   Ligeia
Address:        Secrets Place 3
Phone:  07899921
TESTIING THE CREATION OF A CONTACT ALREADY EXISTING
Contact retrieved:
Name:   Daniele
Address:        Guitars Street 1
Phone:  01234560
Daniele: contact already exists!
TRYING TO RETRIEVE A CONTACT
Foo: contact not found!
UPDATING AN EXISTING CONTACT
Contact stored:
Name:   Daniele
Address:        Mics Street 1
Phone:  07890
REMOVING AN EXISTING CONTACT
Contact Ligeia deleted!
UPDATING AN EXISTING CONTACT
Foo: contact not found!
REMOVING A NON-EXISTENT CONTACT
Ligeia: contact not found!
```

To use the `FileManagerNIO2` class it will be sufficient to specify it as a command line argument instead of `FileManager`.

Solution 19.u)

Thus, we rewrite the constructor of the Exercise19U class:

```
    public Exercise19U(String className) {
        contacts = getContacts();
        fileManager =
          SerializationManagerFactory.getSerializationManager(className);
    }
```

which invokes the static method getSerializationManager() of the SerializationManagerFactory class. In fact, here is the SerializationFactoryManager class:

```
package com.claudiodesio.phonebook.factory;

import com.claudiodesio.phonebook.spi.SerializationManager;
import com.claudiodesio.phonebook.data.Contact;
import java.util.Iterator;
import java.util.ServiceLoader;

public class SerializationManagerFactory {
    public static SerializationManager<Contact> getSerializationManager(
      String className) {
        ServiceLoader<SerializationManager> serviceLoader = ServiceLoader.load(
          com.claudiodesio.phonebook.spi.SerializationManager.class);
        for (SerializationManager serializationManager : serviceLoader) {
            if (serializationManager.getClass().getSimpleName().
                equals(className)) {
                return serializationManager;
            }
        }
        throw new IllegalArgumentException(
          "No serialization manager found for class = " + className);
    }
}
```

The descriptor of the new module com.claudiodesio.phonebook.factory is the following:

```
module com.claudiodesio.phonebook.factory {
    exports com.claudiodesio.phonebook.factory to
      com.claudiodesio.phonebook.test;
    requires com.claudiodesio.phonebook.spi;
    requires com.claudiodesio.phonebook.data;
    uses com.claudiodesio.phonebook.spi.SerializationManager;
}
```

While the descriptor of the module com.claudiodesio.phonebook.test can be simplified as follows:

608

```
module com.claudiodesio.phonebook.test {
    requires com.claudiodesio.phonebook.spi;
    requires com.claudiodesio.phonebook.factory;
    requires com.claudiodesio.phonebook.data;
    requires com.claudiodesio.phonebook.util;
}
```

The command to compile everything will be:

```
javac -d mods --module-source-path src
 src/com.claudiodesio.phonebook.data/module-info.java
 src/com.claudiodesio.phonebook.data/com/claudiodesio/phonebook/data/*.java
 src/com.claudiodesio.phonebook.exceptions/module-info.java
 src/com.claudiodesio.phonebook.exceptions/com/claudiodesio/phonebook/exceptions/*.java
 src/com.claudiodesio.phonebook.util/module-info.java
 src/com.claudiodesio.phonebook.util/com/claudiodesio/phonebook/util/*.java
 src/com.claudiodesio.phonebook.spi/module-info.java
 src/com.claudiodesio.phonebook.spi/com/claudiodesio/phonebook/spi/*.java
 src/com.claudiodesio.phonebook.io/module-info.java
 src/com.claudiodesio.phonebook.io/com/claudiodesio/phonebook/io/*.java
 src/com.claudiodesio.phonebook.nio/module-info.java
 src/com.claudiodesio.phonebook.nio/com/claudiodesio/phonebook/nio/*.java
 src/com.claudiodesio.phonebook.test/module-info.java
 src/com.claudiodesio.phonebook.test/com/claudiodesio/phonebook/test/*.java
 src/com.claudiodesio.phonebook.factory/com/claudiodesio/phonebook/factory/*.java
 src/com.claudiodesio.phonebook.factory/module-info.java
```

The execution command does not change with respect to that used in the previous exercise, as well as the output.

Solution 19.v)

Meanwhile we have renamed the Exercise19U class in Exercise19V and recompiled the modules. We then created the eight modular files with this command:

```
jar --create --file=lib/com.claudiodesio.phonebook.data.jar
 --module-version=1.0 -C mods/com.claudiodesio.phonebook.data .
jar --create --file=lib/com.claudiodesio.phonebook.exceptions.jar
 --module-version=1.0 -C mods/com.claudiodesio.phonebook.exceptions .
jar --create --file=lib/com.claudiodesio.phonebook.util.jar
 --module-version=1.0 -C mods/com.claudiodesio.phonebook.util .
jar --create --file=lib/com.claudiodesio.phonebook.spi.jar
 --module-version=1.0 -C mods/com.claudiodesio.phonebook.spi .
jar --create --file=lib/com.claudiodesio.phonebook.io.jar
 --module-version=1.0 -C mods/com.claudiodesio.phonebook.io .
jar --create --file=lib/com.claudiodesio.phonebook.nio.jar
 --module-version=1.0 -C mods/com.claudiodesio.phonebook.nio .
jar --create --file=lib/com.claudiodesio.phonebook.factory.jar
 --module-version=1.0 -C mods/com.claudiodesio.phonebook.factory .
jar --create --file=lib/com.claudiodesio.phonebook.test.jar
 --module-version=1.0 --main-class = com.claudiodesio.phonebook.test.Exercise19V -C
 mods/com.claudiodesio.phonebook.test .
```

Then we executed the module with the command:

```
java -p lib -m com.claudiodesio.phonebook.test FileManager
```

or alternatively with:

```
java -p lib -m com.claudiodesio.phonebook.test FileManagerNIO2
```

Solution 19.z)

With the following command we create the requested runtime:

```
jlink --module-path "C:/Program Files/Java/jdk-13/jmods"
  --add-modules java.base --output javabasert
```

While with the following command we execute our modular JAR com.claudiodesio.phonebook.test using the created runtime:

```
javabasert\bin\java -p lib -m com.claudiodesio.phonebook.test FileManager
```

As always, just replace FileManager with FileManagerNIO2 to obtain the serialization in the alternative way:

```
javabasert\bin\java -p lib -m com.claudiodesio.phonebook.test FileManagerNIO2
```

Chapter 20
Exercises

Input-Output

The exercises in this chapter will be initially theory oriented, with three "true-false" exercises. Then we will resume the exercise of the phonebook application we made in the first seven chapters by adding the serialization of the objects step by step. We will also create a command line interface using the MVC pattern, creating a considerable degree of complexity, even for the already experienced programmer. As always, then we will find some preparatory exercises for the Oracle Programmer certification.

Exercise 20.a) Input - Output, True or False:

1. The Decorator pattern allows you to implement a sort of dynamic inheritance. This means that, instead of creating as many classes as there are entities to be abstracted, at runtime it will be possible to concretize one of these concepts directly with an object.

2. Readers and writers allow you to read and write characters. For this reason, they are called *Character Stream*.

3. Within the `java.io` package the `Reader` interface has the role of `ConcreteComponent`.

4. Within the `java.io` package the `InputStream` interface has the role of `ConcreteDecorator`.

5. A `BufferedWriter` is a `ConcreteDecorator`.

6. Streams that can communicate directly with a source or destination are called *Node Streams*.

7. The node streams of type OutputStream can use the method:

```
int write (cbuf byte [])
```

to write to a destination.

8. The following in object:

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

allows to use a readLine() method that will read sentences written with the keyboard delimited by the keystroke of the **Enter** key.

9. The following code:

```
File outputFile = new File("foo.txt");
```

creates a text file called **foo.txt** in the current folder.

10. It is not possible to decorate a FileReader.

Exercise 20.b) Serialization, True or False:

1. The state of an object is defined by the value of its instance variables (at a certain time).

2. The Serializable interface has no methods.

3. transient is a modifier applicable to variables and classes. A transient variable is not serialized with the other variables; a transient class is not serializable.

4. transient is a modifier applicable to methods and variables. A transient variable is not serialized with the other variables; a transient method is not serializable.

5. If you tried to serialize an object that has one of its instance variables of type Reader declared as transient, we would get a NotSerializableException at runtime.

6. A static variable will not be serialized.

7. An instance variable of type OuputStream must be declared transient to be involved in the serialization process.

8. It is not possible to use a Serializable implementation as a parameter of a try-with-resources".

9. You can create a clone of a `transient` object using input-output methods.

10. An object can also be serialized in Base64.

Exercise 20.c) New Input Output, True or False:

1. NIO 2.0 completely replaces the input-output model defined with the package `java.io`.

2. The use of the `File` class could be completely replaced by the use of the `Files` class and the `Path` interface.

3. The `toPath()` method of the `File` class returns the equivalent `Path` object.

4. `Path` is an interface because its implementation depends on the platform.

5. The `relativize()` method belongs to the `Files` class and returns the path to get from the `Path` specified as the first argument to the `Path` specified as the second argument.

6. The `subPath()` method of the `Path` interface does not return the root node.

7. The `delete()` method of the `Path` interface will raise an exception if you try to delete a non-empty directory.

8. It makes no sense to get a `Reader` or `Writer` from a `Files` object.

9. The temporary file name is always established by the operating system.

10. A temporary file is saved in a directory that depends on the operating system.

Exercise 20.d)

Create an `InteractiveEditor` class containing a `main()` method that allows you to write in a file what you type from the command line.

Exercise 20.e)

Create a `More` class containing a `main()` method that simulates the program **more** that you can find on Linux operating systems. Once executed, it will wait for the user to specify the input file to read. Once the file is loaded, the program will wait for user input. If the user presses the letter "n" followed by the E**nter** key, then the first 10 lines of the selected file will be displayed, and each time the action is repeated will print the next 10 lines, up to the end of the file. By specifying the letter "q" followed by the E**nter** key the program must end.

Exercise 20.f)

Starting from the solution of the Exercise 7.y, let's continue to evolve our application that simulates the functioning of a phonebook. In this and the next few exercises, we will therefore create new classes to add to our application step by step. Later on, we will integrate the classes created in the Exercise 7.y with those that we will create in the next exercises.

We will use as a "data persistence layer", a component that deals with serializing and deserializing objects on the file system. In other words, we want to create a class called `FileManager`, belonging to the `phonebook.integration` package, and which exposes the following methods:

1. `insert()` which takes an object of type `Contact` as input, and returns nothing. It will have the task of serializing on our file system the `Contact` object that takes as input, in a file with the name of the contact (variable `name` of the `Contact` class) and with suffix **.con**. For example, if the contact name is "Ross", then the file name should be **Ross.con**.

2. `retrieve()` which takes as input the name of the contact to be retrieved through deserialization. For example, if the name "Phoebe Buffay" is passed, the **Phoebe Buffay.con** file must be retrieved.

The only class of the Exercise 7.y that we have to reuse for now should be the `Contact` class, which however extended the `Entity` class, which in turn implemented the `Data` and the `Identifiable` interfaces, so let's start from these files. In order to make the `Contact` class serialization work, we have to make that the `Entity` class also implements the `java.io.Serializable` interface. Also replace the `printDetails()` method of the `Contact` class with a `toString()` method implementation.

Exercise 20.g)

Create an `Exercise20G` test class that verifies that the methods created in the `FileManager` class in the previous exercise are working properly. This class must create an array of three contacts, serialize them and try to reread them.

Exercise 20.h)

Starting from the solution of the previous exercise, create the `Exercise20H` class starting from the `Exercise20G` class by adding (and calling) two methods that test:

1. what happens if you try to add an existing contact;

2. what happens if you try to retrieve a non-existing contact.

614

Think about the test results and:

1. design solutions for any problems that have been encountered;

2. implement the designed solutions.

Exercise 20.i)

Starting from the solution of the previous exercise, let's add the methods in the `FileManager` class:

1. `update()` that modifies a contact (you will not have to be able to change the contact name);

2. `remove()` which deletes the contact with the specified name.

Also create the methods in the test class that check the proper functioning of the `update()` e `remove()` methods.

Exercise 20.j)

Starting from the solution of the previous exercise, let's add another requirement, this time not functional but architectural: applying the dependency inversion principle (DIP, introduced in paragraph 18.2.1). As previously done for the `Contact` class, we therefore evolve the `FileManager` class, using a generalization to represent an implementation of a generic tool to interact with any storage mechanism.

So, let's create a `SerializationManager` interface to be implemented by `FileManager`. This modification will allow us to easily change the way we serialize our objects (see exercise 20.l).

Exercise 20.k)

Starting from the solution of the previous exercise, let's add another requirement. Let's do some code refactoring, removing duplicate code in the created classes. In particular, create the `Exercise20k` class, eliminating the duplications of code that were present in the `Exercise20j` class. Of course, the reader is free to add any improvements to its code in any class.

> **During development, it is very important at some point, to stop, reflect on and enhance your code, by "refactoring", or change the internals structure of the software without changing its external behavior. This may seem like a waste of time to someone, but it is . . .**

> **. . . exactly the opposite. Refactoring improves the quality of the non-functional level code by improving its architectural characteristics. It favors some of what are called "systemic qualities of the software", for example it promotes maintainability and extensibility of the code, reducing its complexity. In this way we will have less bugs and more awareness of our code. My personal advice is to dedicate a daily percentage of the development time that varies from 10% to 25% to refactoring, possibly when you realize you are too tired. For example, on an 8-hour working day, the last two hours (or at least the last) could be dedicated to improving the quality of the software. For information on refactoring you can:**
>
> **- visit** https://refactoring.com
>
> **- buy the "Refactoring" book by Martin Fowler, as recommended in the bibliography of Appendix Z;**
>
> **- use the refactoring mechanisms offered by the most famous IDEs.**

Exercise 20.l)

Starting from the previous Exercise solution, also create an alternative implementation of the `FileManager` class using the NIO2 API (`FileManagerNIO2` class). Also implement the `Exercise20L` test class.

Exercise 20.m)

Starting from the previous exercise, using an object of type `Properties` (introduced in Chapter 14), create a **config.properties** configuration file containing a single property, which our application will exploit to choose whether to use the `FileManager` class or `FileManagerNIO2` class to serialize objects. Create a `SerializationManagerFactory` class, which implements the Factory Method pattern by defining the `getSerializationManager()` method. Finally, create the `Exercise20M` class starting from the `Exercise20L` class, which takes advantage of the mechanism just described for serializing objects.

Exercise 20.n)

> **This and the next exercises are particularly complex. They are hard even for professionals. Sometimes it will be necessary to go on the internet to study libraries and solutions, and be very focused. So even if the results aren't so good, you don't have to worry. If you get stuck, read the solutions and move on. If instead you can solve these exercises in some way, then it means that you have reached a really important programming level!**

Starting with this exercise, we will create a command line user interface. It is a complex exercise that requires a step-by-step procedure. In this first exercise, using an object of type Scanner, we will therefore define a PhonebookCLI class, which represents a user interface from the command line ("CLI" stands for "Command Line Interface"). We will use the Model View Controller (MVC) architectural pattern which is schematized in Figure 20.n.1.
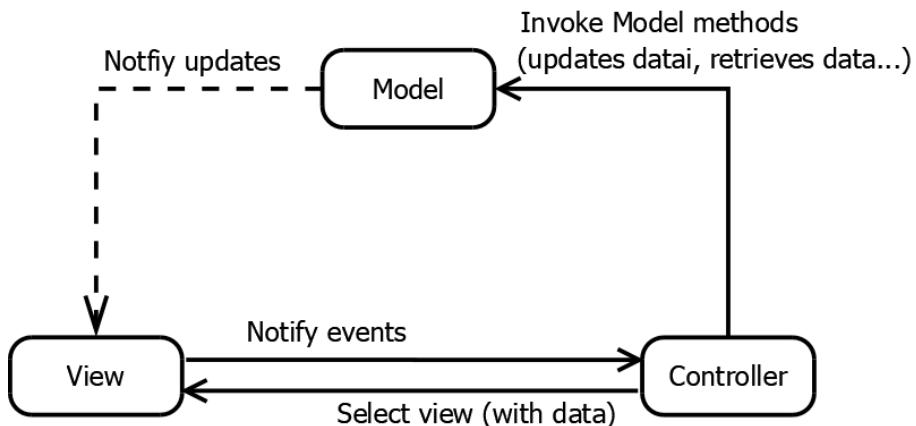
Figure 20.n.1 – Model View Controller.

> **Note that the arrow that goes from the Model to the View has been drawn dashed, because even if it represents a responsibility of the classic MVC pattern, it will not be implemented in our case (as often happens), to make the Model component independent of the View component. The updated Model data will be passed to the View directly from the Controller.**

617

The MVC pattern promotes the separation of responsibilities in a software with different components, defining three main roles, precisely the Model, the View and the Controller. This distinction will guarantee us a series of advantages from the point of view of maintainability, ease of finding bugs, reusability, and so on. We will assume that the role of Model will be interpreted by the set of classes (apart from the test classes) that we have already created and that make up our application so far. The class `PhonebookCLI` instead, will represent the View component, or the component that implements the so-called *presentation logic*. This means that it will have two fundamental responsibilities:

1. define the "screens" for the user (which in our case will consist of simple text);

2. notify the controller component of any user input (implementation that we will do starting from the next exercise).

Also, in our case, we want to emphasize that our `PhonebookCLI` class will have an additional responsibility:

3. handle exceptions (i.e. must use `try-catch` blocks to view any exceptions that have propagated from the Model component).
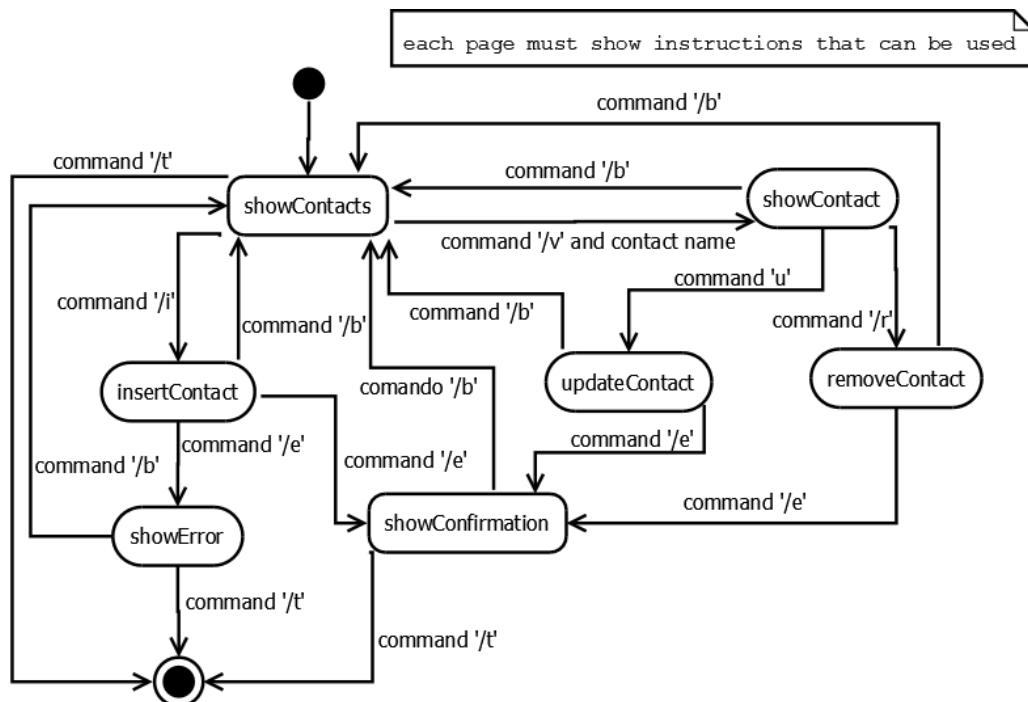
Figure 20.n.2 – State Diagram.

This means that the methods of the Controller and Model components will not handle exceptions but simply allow their propagation. In other words, the methods of the Controller and Model components will not use `try-catch` blocks, but `throws` clauses, to catch exceptions thrown with the `throw` keyword. Our implementation must follow the specifications represented in the state diagram in Figure 20.n.2.

> **The state diagram is usually used to describe the states of an entity (for example an object), which it assumes during its life cycle. It defines the main elements as states (represented as rectangles with rounded edges) and transitions between states. The latter have a verse, and can be labelled with a syntax consisting of three parts (all optional):** `event-trigger [condition] / action`. **Also, there are the start and end elements that we have already encountered with the activity diagram. More information on UML syntax can be found in Appendix G.**

The diagram is quite complex, but it will be enough to take it one step at a time to be able to interpret it, and in the end, we will discover that it is a fundamental specification for our program. In fact, the programming of user interfaces differs a lot from the traditional programming that we have used so far. A diagram like the one specified in Figure 20.n.2 will allow us not to have to improvise a solution.

In the diagram we have used states to represent the "text screens" that must be displayed, and transitions labelled with the event-triggers that lead from one screen to another.

> **Note that we have specified a very specific protocol, where for each screen there is a particular set of commands that can be specified, to move to another screen. All operating commands consist of a single letter preceded by the symbol "/".**

To create the `PhonebookCLI` class, we then begin to create the methods represented in the state diagram. These will have to:

1. print consistent messages

2. capture user input with a `Scanner` object (declare it as an instance variable).

In this exercise, therefore, our tasks must be limited to:

619

1. create the PhonebookCLI class, which has as its instance variable a Scanner object initialized in a constructor.

2. Define the showContacts() method which takes as input a List object of contacts and returns void. For now this method will have to implement the flow described in the diagram, that is, after having printed the list of contacts it takes as input, and the list of possible commands to be specified, wait for user input and with conditional constructs print only (for now) the description of the command typed by the user, then the program can end. For example, if the user types the "/i" command, then the program must print "Called insert contact method". If, on the other hand, the user types the "/v" command followed by the name of a contact in the list (suppose the name "Giovanni"), the program must print "Called method show contact for Giovanni". Finally, if the user specifies the "/t" command, the program must terminate

3. Create an exception called InvalidCommandException that must be thrown in the event that the command specified by the user is not valid.

4. Throw the (already created) NonExistingContactException exception if the user types the command to view the contact "/v" followed by a name that is not in the list.

5. Finally create the Exercise20N class that will be used to launch the application (you must instantiate the PhonebookCLI object and call the method showContacts() by passing it a list of Contact objects as input).

In the next few exercises, we will evolve this class until it becomes fully functional.

> **Starting from this exercise, to capture user input, we strongly recommend using the nextLine() method of the Scanner class, in order to make your life easier. In fact, the joint use of nextLine(), and other methods such as next(), on the same Scanner object, means that the reading of the user input will be managed in a more complex way. So always use only the nextLine() method.**

Exercise 20.o)

Starting from the solution of the previous exercise, we begin to create the PhonebookController class, which will play the role of the Controller component within the MVC architecture. In particular, this class has the task of defining the

so-called *control logic*, that is, it will have to implement the following responsibilities:

1. capture and interpret user inputs notified by the View component;

2. invoke business methods exposed by the Model component (both updating and retrieving data);

3. select the right screen for the View component.

That said, in the previous exercise we created the showContacts() method which was limited to printing certain messages. The time has come to replace those messages with notifications to the Controller component (i.e. the View must call methods on the PhonebookController class). These methods, as we have already said, are represented in the diagram of Figure 20.n.1 as the transitions that lead from one screen to another. PhonebookController methods will always start with the word "handle".

With the previous exercise, we invoked the showContacts() method of the PhonebookCLI class from the Exercise20N class. The Exercise20N class also took care of passing to the method showContacts() the list of contacts. In this exercise, however, the Exercise20O class will invoke a new method of the PhonebookCLI class which we will call start(). This method in turn will invoke the start() method of the PhonebookController class. Here, to for the responsibilities of the Controller component, a method called getContacts() (to be created) of the Model which will return all existing contacts, will be invoked. Once the contacts have been returned by the getContacts() method, the start() method of the PhonebookController class will invoke the showContacts() method of the PhonebookCLI class. All this is summarized in the sequence diagram shown in Figure 20.o.1:
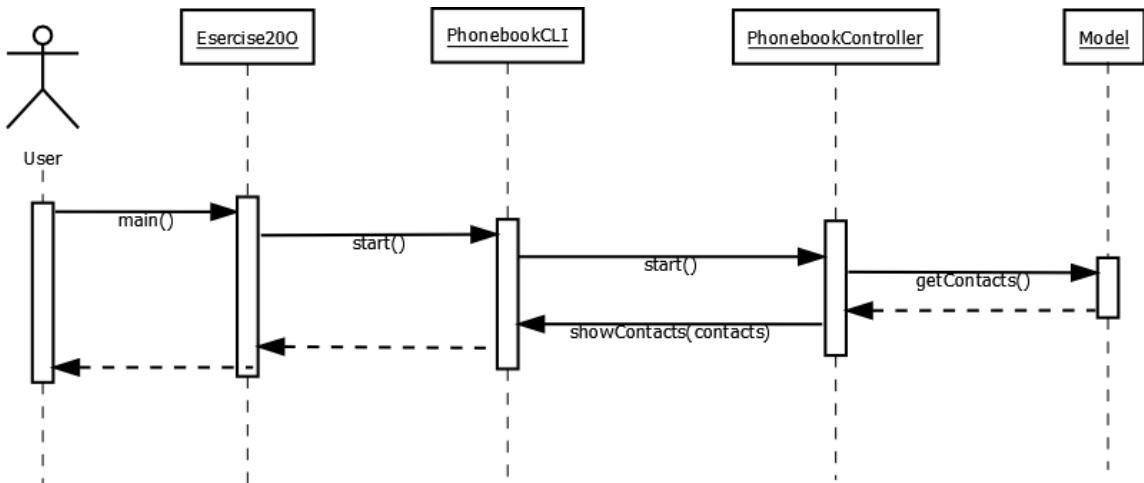
Figure 20.o.1 – Sequence Diagram.

621

So, in this exercise we will have to:

1. Create the `getContacts()` method in the Model component, which returns the contacts already created. Note that initially we will not have contacts to retrieve, so to test the `getContacts()` method, without having to wait for the completion of the next exercise, you can use the **.con** files created in the previous exercises (copying them in your current work directory), or create a test client who inserts some contacts as done in the previous exercises.

2. Create the `PhonebookController` class as a nested class into the `PhonebookCLI` class (in this way we will save several lines of code).

3. We will create the `start()` method in the `PhonebookController` class. This must call the `getContacts()` method of the Model component, and once the contacts are returned, it must pass them to the `PhonebookCLI` `showContacts()` method. In the previous exercise we had invoked the `showContacts()` method of the `PhonebookCLI` class from the `Exercise20N` class. In that case the `Exercise20N` class also took care of creating the list of contacts to be passed to the `showContacts()` method. Now, however, the `start()` method in the `PhonebookController` class will retrieve the contacts and pass them to the `showContacts()` method of the `PhonebookCLI` class.

4. Create the `start()` method in the `PhonebookCLI` class which will invoke the `start()` method of the `PhonebookController` class.

5. Create the `Exercise20O` class that invokes the `start()` method of the `PhonebookCLI` class to start the application.

Exercise 20.p)

Starting from the solution of the previous exercise, implement the insertion flow of a contact as described in the diagram shown in Figure 20.n.2. In particular:

1. Create the `handleShowContacts()` method in the `PhonebookController` class, which must take as input the command entered by the user, and the list of contacts. This method must be invoked in the `showContacts()` method (which must be updated) when the user has typed a command, and must have the responsibility to interpret this command and invoke the appropriate methods. In particular, it must perform the conditional flow that we defined in the `showContacts()` method in the previous exercise, with the exception of the case of insertion. In fact, if the "/i" command has been typed, then the `insertContact()` method of the `PhonebookCLI` class must be invoked. For now, the

rest of the code will be the same as before. In practice, it is necessary to move part of the code that we had in the showContacts() method of the PhonebookCLI class, in the handleShowContacts() method of the PhonebookController class, changing the insertion condition.

2. Change the showContacts() method so that after capturing the user's command, it delegates the interpretation to the handleShowContacts() method of the PhonebookController class, i.e. must invoke this method by passing it the typed command and the list of contacts.

3. Create the insertContact() method in the PhonebookCLI class. This method must request the user, first to enter the name, then the phone number, and finally the address. After collecting the contact information, it must offer the user the opportunity to enter the "/e" command to confirm the entry, or the "/b" command to return to go back to show all the contacts (see Figure 20.n.2). Once the command typed by the user has been retrieved, he must invoke the handleInsertContact() method in the PhonebookController class, which must be responsible for interpreting this command.

4. Create the handleInsertContact() method in the PhonebookController class. This method must take as input the command entered by the user, plus the name, telephone number and address of the contact. This method must have the responsibility to interpret the typed command and must be invoked in the insertContact() method which was responsible for capturing the user input. In particular, if the user has entered the "/b" command, the showContacts() method of the PhonebookCLI class must be invoked again. If the user has entered the "/e" command, it is necessary to invoke the insert() method of the Model component, without handling any exceptions, which instead must be handled directly by the PhonebookCLI methods, as previously mentioned. Finally, it must invoke the showConfirmation() method of the PhonebookCLI class.

5. It must not be possible to create an unnamed contact. Then create an exception called EmptyNameException, to be thrown in the most appropriate method.

6. Create the showConfirmation() method of the PhonebookCLI class. This method must be intended as generic and reusable, must print the phrase "Operation confirmed!", and must take as input the message that it must show immediately after (for example, we could pass the instruction "Inserted contact:\n" + contact.toString()). Finally, he must allow the user to enter the commands described in the diagram in Figure 20.n.2, or the "/b" command to return to the display of the contacts and "/t" to terminate the program. After capturing the command of the user, its interpretation must be delegated to the handleShowMessage() method of the PhonebookController class. The same goes for

623

a generic method called showError(), which can be called in all the catch clauses of the PhonebookCLI methods, to obtain a centralized management of exceptions.

7. The handleShowMessage() method of the PhonebookController class must take as input the command retrieved from the showConfirmation() method of the PhonebookCLI class. Following the diagram in Figure 20.n.2, this method must invoke the showContacts() method if the command sent by the user is "/b", or terminate the program if the command sent by the user is "/t".

8. Create the Exercise20P class starting from the Exercise20Q class to test the application.

<div style="background:#333;color:#fff;padding:4px">Exercise 20.q)</div>

Starting from the solution of the previous exercise, implement the update and removal flows as described in the diagram shown in Figure 20.n.2. In particular:

1. Update the handleShowContacts() in the PhonebookController class, which takes as input the command typed by the user, and the list of contacts. If the user specifies the "/v" command followed by the name of an existing contact, replace the current printing method with the call to the method of a new method: showContact() of the PhonebookCLI class, which takes the contact specified as input.

2. Create the showContact() method of the PhonebookCLI class, which must then show the details of the selected contact and offer the user the possibility to go back to view all the contacts by specifying the command "/b", or specify the command "/u" to update the selected contact, or specify the "/r" command to remove the selected contact. Then he must capture the command and delegate its interpretation to the handleShowContact() method of the PhonebookController class which takes the command and the contact as input.

3. Create the handleShowContact() method which must be responsible for interpreting the command typed. In particular, if the user has entered the "/b" command, the showContacts() method of the PhonebookCLI class must be invoked again. If the user has entered the "/u" command, it is necessary to invoke the updateContact() method of the PhonebookCLI class. If instead the user enters the "/r" command, then the removeContact() method of the PhonebookCLI class must be invoked.

4. Create the updateContact() method of the PhonebookCLI class which must take the selected contact as input. It must ask the user to specify a new phone number, and a new

address. Then indicate to the user the possibility for the user to choose between the "/b" command to return to see all contacts, or the "/e" command to perform the update. Finally, he must capture the user's command and delegate its interpretation to the Controller component, invoking the `handleUpdateContact()` method by passing the command to it, and the name, telephone number and address of the contact.

5. Create the `handleUpdateContact()` method of the `PhonebookController` class to interpret the command to be executed. In particular, in addition to managing the "/b" command as usual to return to see all the contacts in the phonebook, if the user has specified the "/e" command, he must invoke the `update()` method on the Model component, and then invoke the `showConfirmation()` method of the `PhonebookCLI` class.

6. Create the `removeContact()` method of the `PhonebookCLI` class that must capture as input the name of the selected contact. It must indicate to the user the possibility to choose between the "/b" command to return to see all contacts, or the "/e" command to perform the removal of the contact. Finally, it must capture the user's command and delegate its interpretation to the `PhonebookController` class, invoking the `handleRemoveContact()` method by passing to it the command and the contact name.

7. Create the `handleRemoveContact()` method of the `PhonebookController` class to interpret the command to be executed. In particular, in addition to managing the "/b" command as usual to return to see all the contacts in the phonebook, if the user has specified the "/e" command, he must invoke the `remove()` method on the Model component, and then invoke the `showConfirmation()` method of the `PhonebookCLI` class.

8. Create the `Exercise20Q` class starting from the `Exercise20P` class to test the application.

Exercise 20.r)

Which of the following statements are correct?

1. The `File` class defines a `getPath()` method which returns a string.

2. The `File` class defines a `getParent()` method that returns a `File` object representing a directory.

3. `pathSeparator` is a static constant of the `File` class, which represents the path separator dependent on the operating system.

4. The `File` class defines a `delete()` method that returns a boolean `true` if the file will actually be deleted, or `false` otherwise.

625

Exercise 20.s)

Which of the following statements are correct?

1. `InputStream` implementations are optimized for reading bytes, `OutputStream` implementations are optimized for writing bytes. In fact, they are called "byte streams".

2. The classes at the base of the character stream hierarchy are `Reader` and `Writer`.

3. The `readLine()` method is defined in the `Reader` class.

4. The constructor of an `ObjectInputStream` must take an object of type `FileInputStream` as input.

Exercise 20.t)

Which of the following statements are correct?

1. If we instantiate a `File` object by specifying the name of a file that does not exist as a constructor parameter, it will be created.

2. To read the contents of a text file, just use the `FileReader` class.

3. A directory can be created thanks to the `File` class.

4. The `File` class defines a method that returns file names within a directory.

Exercise 20.u)

Given the following class:

```
import java.io.*;

public class Exercise20U {
    public static void main(String args[]) throws IOException {
        try (FileOutputStream fos = new FileOutputStream("new file.txt");
             DataOutputStream dos = new DataOutputStream(fos);) {
            dos.writeInt(8);
            dos.writeDouble(0.1176);
        }
    }
}
```

Once the **new file.txt** file has been created, how much space it will occupy on the hard disk?

1. No file will be created because the file is not closed correctly.

2. 12 bytes.

3. No file will be created because the file does not compile.

4. 64 bytes.

5. No file will be created because an exception will be thrown during execution.

6. 128 bytes.

7. The size of the file depends on the platform on which the program is run.

8. 96 bytes.

Exercise 20.v)

Given the following class:

```java
import java.io.*;

public class Exercise20S {
    public static void main(String args[]) throws Exception {
        try (FileOutputStream fos = new FileOutputStream("nuovo file.txt");
             DataOutputStream dos = new DataOutputStream(fos);) {
            for (int i = 0; i < 100; i++) {
                dos.writeInt(i);
            }
        }
    }
}
```

Write the class that reads the created file.

Exercise 20.w)

Which of the following statements are correct?

1. To instantiate a Reader we have to handle an IOException.

2. To write from an OutputStream we have to handle an IOException.

3. To read from an InputStream object we have to handle an IOException.

4. To instantiate a Writer object we have to handle a FileNotFoundException

Exercise 20.x)

Which of the following statements are correct?

1. It is important to close a stream, otherwise the memory it occupies will not be deallo-cated until the end of the program.

2. The only class that provides Java to read command-line instructions interactively is the Scanner class.

3. The BufferedReader class declares a method called readLine() which reads a whole line of text from the source passed in input to the BufferedReader.

4. The BufferedReader class declares a method called lines() which returns an object

5. Stream<String>.

Exercise 20.y)

Which of the following statements are correct?

1. Path defines a method that returns the names of the files within a directory.

2. Path declares a method to return the directory in which it is contained.

3. Path declares a method to return the author (owner) of the file.

4. Path is an interface.

Exercise 20.z)

Which of the following statements are correct?

1. The Files class defines a method that returns the names of files within a directory.

2. The find() method of the Files class returns a Stream of Path objects.

3. Files and Path belong to the java.nio2.file package.

4. Files is an interface.

5. You can copy one file to another with the copy() method.

Exercise 20.aa) Networking, True or False:

1. At least two sockets must exist in a network communication.

2. To connect to a server, a client must know at least its IP address and the port on which it has been listening.

3. A server can also be listened to on port 80, the default port of HTTP, without necessarily using that protocol. In fact, it is also possible to communicate with the HTTP protocol on a port other than 80.

4. The `accept()` method locks the server into a "waiting for connections" state. When a client connects, the `accept()` method is executed to collect all client information in an object of type `Socket`.

5. A `ServerSocket` does not need to declare the IP address, but only has to declare the port on which it will wait for connections.

Chapter 20
Exercise Solutions

Input-Output

1. True.
2. True.
3. False.
4. False.
5. True.
6. True.
7. True.
8. True.
9. False.
10. False.

Solution 20.b) Serialization, True or False:

1. True.
2. True.

3. **False**.

4. **False**.

5. **False**.

6. **True**.

7. **True**.

8. **False**.

9. **True**.

10. **True**.

Solution 20.c) New Input Output, True or False:

1. **True**.

2. **True**.

3. **True**.

4. **True**.

5. **False**.

6. **True**.

7. **True**.

8. **False**.

9. **False**.

10. **True**.

Solution 20.d)

The code should be similar to the following:

```
package com.claudiodesio.io;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;
```

```
public class InteractiveEditor {

    public static void main(String args[]) {
        File file = new File("file.txt");
        try (Scanner scanner = new Scanner(System.in);
                FileWriter fileWriter = new FileWriter(file);) {
            String string = "";
            System.out.println("Type something and type enter, " +
                                "or type \"end\" to terminate the program");
            while (!(string = scanner.nextLine()).equals("end")) {
                System.out.println("You typed " + string);
                fileWriter.append(string);
                fileWriter.flush();
            }

        } catch (IOException ex) {
            ex.printStackTrace();
        }
        System.out.println("Program terminated");
    }
}
```

Solution 20.e)

The code should be similar to the following:

```
package com.claudiodesio.io;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class More {

    private final static int numberOfLines = 10;
    private static final String QUIT_COMMAND = "q";
    private static final String NEXT_COMMAND = "n";

    public static void main(String args[]) {
        String character;
        System.out.println("Type the name of a file and type enter, or "
            + "type \"" + QUIT_COMMAND + "\" to terminate the program");
        try (Scanner in = new Scanner(System.in);) {
            String fileName;
            if (!(fileName = in.nextLine()).equals(QUIT_COMMAND)) {
                File file = new File(fileName);
                try (Scanner fileScanner = new Scanner(file)) {
                    System.out.println("File found! \"" + fileName +
```

633

```
                              "\" , press \"n\" to see the next 10 lines");
                     while (!(character = in.nextLine()).equals(QUIT_COMMAND)) {
                         if (character.equals(NEXT_COMMAND)) {
                             for (int i = 0; fileScanner.hasNext() &&
                                 i < numberOfLines; i++) {
                                 System.out.println(fileScanner.nextLine());
                             }
                         }
                     }
                 } catch (FileNotFoundException exc) {
                     System.out.println("File not found!");
                 }
             }
         } catch (Exception ex) {
             ex.printStackTrace();
         }
         System.out.println("Program terminated!");
     }
 }
```

Below is an example of output:

```
java com.claudiodesio.io.More

Type the name of a file and type enter, or type "q" to terminate the program
More.java
File found! "More.java", press "n" to see the next 10 lines
n
package com.claudiodesio.io;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class More {

    private final static int numberOfLines = 10;
    private static final String QUIT_COMMAND = "q";
n
    private static final String NEXT_COMMAND = "n";

    public static void main(String args[]) {
        String character;
        System.out.println("Type the name of a file and type enter, or "
          + "type \"" + QUIT_COMMAND + "\" to terminate the program");
        try (Scanner in = new Scanner(System.in);) {
            String fileName;
            if (!(fileName = in.nextLine()).equals(QUIT_COMMAND)) {
                File file = new File(fileName);
n
                try (Scanner fileScanner = new Scanner(file)) {
                    System.out.println("File found! \"" + fileName
```

```
                      + "\", press \"n\" to see the next 10 lines");
                   while (!(character = in.nextLine()).equals(QUIT_COMMAND)) {
                       if (character.equals(NEXT_COMMAND)) {
                           for (int i = 0; fileScanner.hasNext() &&
                               i < numberOfLines; i++) {
                               System.out.println(fileScanner.nextLine());
                           }
                       }
n
                   }
               } catch (FileNotFoundException exc) {
                   System.out.println("File not found!" + fileName);
               }
           }
       } catch (Exception ex) {
           ex.printStackTrace();
       }
       System.out.println("Program terminated!");
    }
n
}
q
Program terminated!
```

Solution 20.f)

We decided to define a FileUtils utility class defined as follows:

```
package phonebook.util;

public class FileUtils {

    public static final String SUFFIX = ".con";

    public static String getFileName(String name) {
        return name + SUFFIX;
    }
}
```

Then we have decided to implement the FileManager class, which takes care of implementing the two required methods, insert() and retrieve(), in the following way:

```
package phonebook.integration;
import java.util.*;
import java.io.*;
import phonebook.util.*;
import phonebook.data.*;

public class FileManager {
```

```
     public void insert(Contact contact) throws IOException {
         try (FileOutputStream fos =
         new FileOutputStream (new File(FileUtils.getFileName(
                                             contact.getName()))));
         ObjectOutputStream s = new ObjectOutputStream (fos);) {
             s.writeObject (contact);
         }
     }

     public Contact retrieve(String name) throws IOException,
                                         ClassNotFoundException {
         Contact contact = null;
         try (FileInputStream fis = new FileInputStream(
                                     new File(name + FileUtils.SUFFIX));
         ObjectInputStream ois = new ObjectInputStream (fis);) {
             contact = (Contact)ois.readObject();
         }
         return contact;
     }
 }
```

Note that we used the `try`-with-resources construct in both methods.

Solution 20.g)

The required test class follows:

```
package phonebook.test ;

import phonebook.data.*;
import phonebook.integration.*;

public class Exercise20G {

    private FileManager fileManager;
    private Contact[] contacts;

    Exercise20G() {
        contacts = getContacts();
        fileManager = new FileManager();
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS...");
        retrieveContacts();
    }
```

```java
    public void createContacts() {
        for (Contact contact : contacts) {
            createContact(contact);
        }
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            retrieveContact(contact.getName());
        }
    }

    public void retrieveContact(String contactName) {
        try {
            Contact contact = fileManager.retrieve(contactName);
            System.out.println("Retrieved contact:\n" + contact);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    private void createContact(Contact contact) {
        try {
            fileManager.insert(contact);
            System.out.println("Created contact:\n" + contact);
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }

    private Contact[] getContacts() {
        Contact contact1 = new Contact("Daniele", "01234560",
                                       "Guitars Street 1");
        Contact contact2 = new Contact("Giovanni", "0565432190",
                                       "Sciences Square 2");
        Contact contact3 = new Contact("Ligeia", "07899921",
                                       "Secrets Avenue 3");
        Contact[] contacts = {
            contact1, contact2, contact3
        } ;
        return contacts;
    }

    public static void main(String args[]) {
        Exercise20G exercise20G = new Exercise20G();
        exercise20G.executeTest();
    }
}
```

that will produce the following output:

```
TESTING THE CREATION OF THE THREE CONTACTS...
Created contact:
Name: Daniele
Phone:       01234560
Address:     Guitars Street 1
Created contact:
Name: Giovanni
Phone:       0565432190
Address:     Sciences Square 2
Created contact:
Name: Ligeia
Phone:       07899921
Address:     Secrets Avenue 3
RETRIEVING THE THREE CONTACTS...
Retrieved contact:
Name: Daniele
Phone:       01234560
Address:     Guitars Street 1
Retrieved contact:
Name: Giovanni
Phone:       0565432190
Address:     Sciences Square 2
Retrieved contact:
Name: Ligeia
Phone:       07899921
Address:     Secrets Avenue 3
```

Solution 20.h)

We could code the new methods as in the following file (in bold the new code):

```java
package phonebook.test;

import phonebook.exceptions.*;
import phonebook.integration.*;
import phonebook.data.*;

public class Exercise20H {
    private FileManager fileManager;
    private Contact[] contacts;

    Exercise20H() {
        contacts = getContacts();
        fileManager = new FileManager();
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
```

638

```
    System.out.println("RETRIEVING THE THREE CONTACTS...");
    retrieveContacts();
    System.out.println(
      "TESTING THE CREATION OF AN ALREADY EXISTING CONTACT");
    createExistingContact();
    System.out.println("TRYING TO RECOVER A NON-EXISTING CONTACT");
    retrieveNonExistingContact();
}

public void createExistingContact() {
    createContact(contacts[0]);
}

public void retrieveNonExistingContact() {
    retrieveContact("Foo");
}

public void createContacts() {
    for (Contact contact : contacts) {
        createContact(contact);
    }
}

public void retrieveContacts() {
    for (Contact contact : contacts) {
        retrieveContact(contact.getName());
    }
}

public void retrieveContact(String contactName) {
    try {
        Contact contact = fileManager.retrieve(contactName);
        System.out.println("Retrieved contact:\n" + contact);
    }
    catch (Exception exc) {
        exc.printStackTrace();
    }
}

private void createContact(Contact contact) {
    try {
        fileManager.insert(contact);
        System.out.println("Created contact:\n" + contact);
    }
    catch (Exception exc) {
        exc.printStackTrace();
    }
}
```

```
      private Contact[] getContacts() {
          Contact contact1 = new Contact("Daniele", "01234560",
                                         "Guitars Street 1");
          Contact contact2 = new Contact("Giovanni", "0565432190",
                                         "Sciences Square 2");
          Contact contact3 = new Contact("Ligeia", "07899921",
                                         "Secrets Avenue 3");
          Contact[] contacts = {
              contact1, contact2, contact3
          };
          return contacts;
      }

      public static void main(String args[]) {
          Exercise20H exercise20H = new Exercise20H();
          exercise20H.executeTest();
      }
  }
```

The output will be:

```
TESTING THE CREATION OF THE THREE CONTACTS...
Created contact:
Name:    Daniele
Phone:        01234560
Address:  Guitars Street 1
Created contact:
Name:    Giovanni
Phone:        0565432190
Address:  Sciences Square 2
Created contact:
Name:    Ligeia
Phone:        07899921
Address:  Secrets Avenue 3
RETRIEVING THE THREE CONTACTS...
Retrieved contact:
Name:    Daniele
Phone:        01234560
Address:  Guitars Street 1
Retrieved contact:
Name:    Giovanni
Phone:        0565432190
Address:  Sciences Square 2
Retrieved contact:
Name:    Ligeia
Phone:        07899921
Address:  Secrets Avenue 3
TESTING THE CREATION OF AN ALREADY EXISTING CONTACT
Created contact:
Name:    Daniele
Phone:        01234560
Address:  Guitars Street 1
TRYING TO RECOVER A NON-EXISTING CONTACT
```

640

```
java.io.FileNotFoundException: Foo.con (Impossibile trovare il file specificato)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:213)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:155)
        at phonebook.integration.FileManager.retrieve(FileManager.java:52)
        at phonebook.test.Exercise20H.retrieveContact(Exercise20H.java:49)
        at phonebook.test.Exercise20H.retrieveNonExistingContact(Exercise20H.java:32)
        at phonebook.test.Exercise20H.executeTest(Exercise20H.java:24)
        at phonebook.test.Exercise20H.main(Exercise20H.java:79)
```

Note that the test showed that:

1. adding an existing contact causes the old contact to be overwritten;

2. recovering a non-existent contact causes a `FileNotFoundException`.

We are not satisfied with both test results. In the first case, it does not seem correct to us that the old contact has been replaced without the user even being aware of it. We decided to implement a simple solution that always check if a contact already exists before serialize it, and in the case throw a custom exception that could be called `DuplicateContactException`.

In the second case, it would be preferable to create a custom exception that we could call `NonExistingContactException`, containing a readable message.

Let's now implement the solution.

First, let's define the exceptions in a simple way. Here it is the `NonExistingContactException` class:

```
package phonebook.exceptions;

import java.io.IOException;

public class NonExistingContactException extends IOException {

    private static final long serialVersionUID = 89424022400056525663L;

    public NonExistingContactException(String message) {
        super(message);
    }
}
```

And then follows the `ContattoEsistenteException` class:

```
package phonebook.exceptions;

import java.io.IOException;

public class DuplicateContactException extends IOException {
```

```
        private static final long serialVersionUID = 8942402240056525662L;

        public DuplicateContactException(String message) {
            super(message);
        }
    }
```

Note that we both classes extend the IOException class. So, let's modify the FileManager class in the following way:

```
package phonebook.integration;

import java.util.*;
import java.io.*;
import phonebook.exceptions.*;
import phonebook.util.*;
import phonebook.data.*;

public class FileManager {
    public void insert(Contact contact) throws DuplicateContactException,
                                        FileNotFoundException, IOException {
        Contact existingContact = getContact(
                                    FileUtils.getFileName(contact.getName()));
        if (existingContact != null) {
            throw new DuplicateContactException(contact.getName()
                + ": contact already existing!");
        }
        try (FileOutputStream fos = new FileOutputStream (
        new File(FileUtils.getFileName(contact.getName()))));
        ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contact);
        }
    }

    public Contact retrieve(String name) throws NonExistingContactException {
        Contact contact = getContact(FileUtils.getFileName(name));
        if (contact == null) {
            throw new NonExistingContactException(
                name + ": contact not found!");
        }
        return contact;
    }

    private Contact getContact(String name) {
        try (FileInputStream fis = new FileInputStream(new File(name));
        ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contact contact = (Contact)ois.readObject();
            return contact;
        } catch (Exception exc) {
```

```
            return null;
        }
    }
}
```

Note that we have added the private method `getContact()`, which returns the contact if it exists, or `null` if it is not found. This method is used by both the `insert()` and the `retrieve()` method, in order to handle any exceptions. Finally, we also modified the test class by replacing, with the printing of the `getMessage()` method of the exceptions, the statement that printed the stack trace of the exception (see the bold statements):

```
package phonebook.test;

import phonebook.exceptions.*;
import phonebook.integration.*;
import phonebook.data.*;

public class Exercise20H {
    private FileManager fileManager;
    private Contact[] contacts;

    Exercise20H() {
        contacts = getContacts();
        fileManager = new FileManager();
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS...");
        retrieveContacts();
        System.out.println("TESTING THE CREATION OF AN ALREADY EXISTING CONTACT");
        createExistingContact();
        System.out.println("TRYING TO RECOVER A NON-EXISTING CONTACT");
        retrieveNonExistingContact();
    }

    public void createExistingContact() {
        createContact(contacts[0]);
    }

    public void retrieveNonExistingContact() {
        retrieveContact("Foo");
    }

    public void createContacts() {
        for (Contact contact : contacts) {
```

```
                createContact(contact);
        }
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            retrieveContact(contact.getName());
        }
    }

    public void retrieveContact(String contactName) {
        try {
            Contact contact = fileManager.retrieve(contactName);
            System.out.println("Retrieved contact:\n" + contact);
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private void createContact(Contact contact) {
        try {
            fileManager.insert(contact);
            System.out.println("Created contact:\n" + contact);
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private Contact[] getContacts() {
        Contact contact1 = new Contact("Daniele", "01234560",
                                    "Guitars Street 1");
        Contact contact2 = new Contact("Giovanni", "0565432190",
                                    "Sciences Square 2");
        Contact contact3 = new Contact("Ligeia", "07899921",
                                    "Secrets Avenue 3");
        Contact[] contacts = {
            contact1, contact2, contact3
        } ;
        return contacts;
    }

    public static void main(String args[]) {
        Exercise20H exercise20H = new Exercise20H();
        exercise20H.executeTest();
    }
}
```

644

> **Delete the** .con **files generated by the previous execution of the test, in order not to get false results in the test.**

The output will be:

```
TESTING THE CREATION OF THE THREE CONTACTS...
Created contact:
Name: Daniele
Address:      Guitars Street 1
Phone:        01234560
Created contact:
Name: Giovanni
Address:      Sciences Square 2
Phone:        0565432190
Created contact:
Name: Ligeia
Address:      Secrets Avenue 3
Phone:        07899921
RETRIEVING THE THREE CONTACTS...
Retrieved contact:
Name: Daniele
Address:      Guitars Street 1
Phone:        01234560
Retrieved contact:
Name: Giovanni
Address:      Sciences Square 2
Phone:        0565432190
Retrieved contact:
Name: Ligeia
Address:      Secrets Avenue 3
Phone:        07899921
TESTING THE CREATION OF AN ALREADY EXISTING CONTACT
Daniele: contact already existing!
TRYING TO RECOVER A NON-EXISTING CONTACT
Foo: contact not found!
```

Solution 20.i)

Consistent with the development line held so far, the FileManager file could evolve as follows (changes in bold):

```
package phonebook.integration;

import java.util.*;
import java.io.*;
import phonebook.exceptions.*;
import phonebook.util.*;
import phonebook.data.*;
```

```
public class FileManager {

    public void insert(Contact contact) throws DuplicateContactException,
                                    FileNotFoundException, IOException {
        Contact existingContact = getContact(
                             FileUtils.getFileName(contact.getName()));
        if (existingContact != null) {
            throw new DuplicateContactException(contact.getName() +
                ": contact already existing!");
        }
        register(contact);
    }

    public Contact retrieve(String name) throws NonExistingContactException {
        Contact contact = getContact(FileUtils.getFileName(name));
        if (contact == null) {
            throw new NonExistingContactException(name
                                                + ": contact not found!");
        }
        return contact;
    }

    public void update(Contact contact) throws NonExistingContactException,
      DuplicateContactException, FileNotFoundException, IOException {
        if (isExistingContact(contact.getName())) {
            register(contact);
        } else {
            throw new NonExistingContactException(contact.getName() +
                ": contact not found!");
        }
    }

    public void remove(String name) throws NonExistingContactException,
      DuplicateContactException, FileNotFoundException, IOException {
        File file = new File(FileUtils.getFileName(name));
        if (!file.delete()) {
            throw new NonExistingContactException(name
                                                + ": contact not found!");
        }
    }

    private void register(Contact contact) throws FileNotFoundException,
                                                IOException {
        try (FileOutputStream fos = new FileOutputStream (
            new File(FileUtils.getFileName(contact.getName())));
        ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contact);
        }
    }
```

```
    private boolean isExistingContact(String name) {
        File file = new File(FileUtils.getFileName(name));
        return file.exists();
    }

    private Contact getContact(String name) {
        try (FileInputStream fis = new FileInputStream(new File(name));
        ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contact contact = (Contact)ois.readObject();
            return contact;
        } catch (Exception exc) {
            return null;
        }
    }
}
```

Note that the private method isExistingContact() only checks if a file with the contact name specified exists, and this in our case should be sufficient. We invoke this method from the update() method before recording (and overwriting) the old contact. If the file does not exist, we would get a NonExistingContactException. The remove() method instead takes advantage of the delete() method of the File class. This returns true if and only if it successful delete the file. We could modify the test class in the following way, to test the modification and cancellation scenarios (changes in bold):

```
package phonebook.test;

import phonebook.exceptions.*;
import phonebook.integration.*;
import phonebook.data.*;


public class Exercise20I {

    private FileManager fileManager;

    private Contact[] contacts;

    Exercise20I() {
        contacts = getContacts();
        fileManager = new FileManager();
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS...");
        retrieveContacts();
```

```
        System.out.println(
            "TESTING THE CREATION OF A ALREADY EXISTING CONTACT");
        createExistingContact();
        System.out.println("TRYING TO RECOVER A NON-EXISTING CONTACT");
        retrieveNonExistingContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateExistingContact();
        System.out.println("REMOVING AN EXISTING CONTACT");
        removeExistingContact();
        System.out.println("UPDATING A NON-EXISTING CONTACT");
        updateNonExistingContact();
        System.out.println("REMOVING A NON-EXISTING CONTACT");
        removeNonExistingContact();
    }

    public void createContacts() {
        for (Contact contact : contacts) {
            createContact(contact);
        }
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            retrieveContact(contact.getName());
        }
    }

    public void createExistingContact() {
        createContact(contacts[0]);
    }

    public void retrieveNonExistingContact() {
        retrieveContact("Foo");
    }

    public void updateExistingContact() {
        try {
            Contact contact = new Contact("Daniele","Mics Street 1","07890");
            fileManager.update(contact);
            System.out.println("Contact " + contact.getName()
                                + " updated!\n" + contact);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void removeExistingContact() {
        try {
            fileManager.remove(contacts[2].getName());
            System.out.println("Contact " + contacts[2].getName()
                                + " removed!");
```

```java
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void updateNonExistingContact() {
        try {
            Contact contact = new Contact("Bar","Mics Street 1","07890");
            fileManager.update(contact);
            System.out.println("Contact " + contact.getName()
                            + " updated!");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void removeNonExistingContact() {
        try {
            String name = "Ligeia";
            fileManager.remove(name);
            System.out.println("Contact " + name + " removed!");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void retrieveContact(String contactName) {
        try {
            Contact contact = fileManager.retrieve(contactName);
            System.out.println("Retrieved contact:\n" + contact);
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private void createContact(Contact contact) {
        try {
            fileManager.insert(contact);
            System.out.println("Created contact:\n" + contact);
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private Contact[] getContacts() {
        Contact contact1 = new Contact("Daniele", "01234560",
                                "Guitars Street 1");
```

649

```
            Contact contact2 = new Contact("Giovanni", "0565432190",
                                            "Sciences Square 2");
            Contact contact3 = new Contact("Ligeia", "07899921",
                                            "Secrets Avenue 3");
            Contact[] contacts = {
                contact1, contact2, contact3
            };
            return contacts;
        }
        public static void main(String args[]) {
            Exercise20I exercise20I = new Exercise20I();
            exercise20I.executeTest();
        }
    }
```

The output will be:

```
TESTING THE CREATION OF THE THREE CONTACTS...
Created contact:
Name: Daniele
Address:     Guitars Street 1
Phone:       01234560
Created contact:
Name: Giovanni
Address:     Sciences Square 2
Phone:       0565432190
Created contact:
Name: Ligeia
Address:     Secrets Avenue 3
Phone:       07899921
RETRIEVING THE THREE CONTACTS...
Retrieved contact:
Name: Daniele
Address:     Guitars Street 1
Phone:       01234560
Retrieved contact:
Name: Giovanni
Address:     Sciences Square 2
Phone:       0565432190
Retrieved contact:
Name: Ligeia
Address:     Secrets Avenue 3
Phone:       07899921
TESTING THE CREATION OF AN ALREADY EXISTING CONTACT
Daniele: contact already existing!
TRYING TO RECOVER A NON-EXISTING CONTACT
Foo: contact not found!
UPDATING AN EXISTING CONTACT
Contact Daniele updated!
Name: Daniele
Address:     07890
Phone:       Mics Street 1
REMOVING AN EXISTING CONTACT
```

```
Contact Ligeia removed!
UPDATING A NON-EXISTING CONTACT
Bar: contact not found!
REMOVING A NON-EXISTING CONTACT
Ligeia: contact not found!
```

Solution 20.j)

We could create the following SerializationManager abstract class:

```java
import java.io.*;

import java.util.*;

public interface SerializationManager<T extends Data> {

    void insert(T data) throws IOException;

    T retrieve(String id) throws IOException, ClassNotFoundException;

    void update(T data) throws IOException;

    void remove(String id) throws IOException;
}
```

to have it implemented by FileManager:

```java
import java.util.*;
import java.io.*;

public class FileManager implements SerializationManager<Contact>  {

    @Override
    public void insert(Contact contact) throws DuplicateContactException,
                                        FileNotFoundException, IOException {
        Contact existingContact = getContact(
                                FileUtils.getFileName(contact.getName()));
        if (existingContact != null) {
            throw new DuplicateContactException(contact.getName() +
                ": contact already existing!");
        }
        store(contact);
    }

    @Override
    public Contact retrieve(String name) throws NonExistingContactException {
        Contact contact = getContact(FileUtils.getFileName(name));
        if (contact == null) {
```

```java
                throw new NonExistingContactException(name
                                              + ": contact not found!");
        }
        return contact;
    }

    @Override
    public void update(Contact contact) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        if (isExistingContact(contact.getName())) {
            store(contact);
        } else {
            throw new NonExistingContactException(contact.getName() +
                ": contact not found!");
        }
    }

    @Override
    public void remove(String name) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        File file = new File(FileUtils.getFileName(name));
        if (file.delete()) {
            System.out.println("Contact " + name + " deleted!");
        } else {
            throw new NonExistingContactException(name + ": contact not found!");
        }
    }

    private void store(Contact contact) throws  FileNotFoundException,
                                                IOException {
        try (FileOutputStream fos =
        new FileOutputStream (new File(FileUtils.getFileName(
                contact.getName()))));
        ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contact);
            System.out.println("Contact storeed:\n" + contact);
        }
    }

    private boolean isExistingContact(String name) {
        File file = new File(FileUtils.getFileName(name));
        return file.exists();
    }

    private Contact getContact(String name) {
        try (FileInputStream fis = new FileInputStream (new File(name));
             ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contact contact = (Contact)ois.readObject();
            System.out.println("Contact retrieved:\n" + contact);
```

652

```
            return contact;
        } catch (Exception exc) {
            return null;
        }
    }
}
```

Finally, the test class follows:

```
package phonebook.test;

import phonebook.exceptions.*;
import phonebook.integration.*;
import phonebook.data.*;

public class Exercise20J {

    private SerializationManager<Contact> fileManager;

    private Contact[] contacts;

    Exercise20J() {
        contacts = getContacts();
        fileManager = new FileManager();
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS...");
        retrieveContacts();
        System.out.println("TESTING THE CREATION OF A ALREADY EXISTING CONTACT");
        createExistingContact();
        System.out.println("TRYING TO RECOVER A NON-EXISTING CONTACT");
        retrieveNonExistingContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateExistingContact();
        System.out.println("REMOVING AN EXISTING CONTACT");
        removeExistingContact();
        System.out.println("UPDATING A NON-EXISTING CONTACT");
        updateNonExistingContact();
        System.out.println("REMOVING A NON-EXISTING CONTACT");
        removeNonExistingContact();
    }

    public void createContacts() {
        for (Contact contact : contacts) {
            createContact(contact);
        }
    }
```

```java
    public void retrieveContacts() {
        for (Contact contact : contacts) {
            retrieveContact(contact.getName());
        }
    }

    public void createExistingContact() {
        createContact(contacts[0]);
    }

    public void retrieveNonExistingContact() {
        retrieveContact("Foo");
    }

    public void updateExistingContact() {
        try {
            Contact contact = new Contact("Daniele","Mics Street 1","07890");
            fileManager.update(contact);
            System.out.println("Contact " + contact.getName() + " updated!\n"
                               + contact);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void removeExistingContact() {
        try {
            fileManager.remove(contacts[2].getName());
            System.out.println("Contact " + contacts[2].getName() + " removed!");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void updateNonExistingContact() {
        try {
            Contact contact = new Contact("Bar","Mics Street 1","07890");
            fileManager.update(contact);
            System.out.println("Contact " + contact.getName() + " updated!");
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void removeNonExistingContact() {
        try {
            String name = "Ligeia";
            fileManager.remove(name);
            System.out.println("Contact " + name + " removed!");
```

```
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void retrieveContact(String contactName) {
        try {
            Contact contact = fileManager.retrieve(contactName);
            System.out.println("Retrieved contact:\n" + contact);
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private void createContact(Contact contact) {
        try {
            fileManager.insert(contact);
            System.out.println("Created contact:\n" + contact);
        }
        catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private Contact[] getContacts() {
        Contact contact1 = new Contact("Daniele", "01234560",
            "Guitars Street 1");
        Contact contact2 = new Contact("Giovanni", "0565432190",
            "Sciences Square 2");
        Contact contact3 = new Contact("Ligeia", "07899921",
            "Secrets Avenue 3");
        Contact[] contacts = {
            contact1, contact2, contact3
        } ;
        return contacts;
    }

    public static void main(String args[]) {
        Exercise20J exercise20J = new Exercise20J();
        exercise20J.executeTest();
    }
}
```

Solution 20.k)

A solution would be to use lambda expressions to avoid rewriting the various methods while always handling exceptions in the same way. In particular, we can create two types of functional

interfaces. One that defines a method that returns `void`:

```
package phonebook.util;

@FunctionalInterface
public interface Executor {

    void execute() throws Exception;
}
```

And another that returns a generic type:

```
package phonebook.util;

@FunctionalInterface
public interface Retriever<O> {

    O execute() throws Exception;
}
```

> **We did not use a** `java.util.function.Consumer` **functional interface in place of** `Executor`, **and the** `java.util.function.Supplier` **functional interface in place of** `Retriever`, **because we needed the SAM methods to declare the** `throws` **clause to** `Exception`

We can exploit these functional interfaces in the `Exercise20K` class in the following way (the parts modified as usual are in bold):

```
package phonebook.test;

import phonebook.exceptions.*;
import phonebook.integration.*;
import phonebook.data.*;
import phonebook.util.*;

public class Exercise20K {

    private SerializationManager<Contact> fileManager;

    private Contact[] contacts;

    Exercise20K() {
        contacts = getContacts();
        fileManager = new FileManager();
    }
```

```
    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS...");
        retrieveContacts();
        System.out.println("TESTING THE CREATION OF A ALREADY EXISTING CONTACT");
        createExistingContact();
        System.out.println("TRYING TO RECOVER A NON-EXISTING CONTACT");
        retrieveNonExistingContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateExistingContact();
        System.out.println("REMOVING AN EXISTING CONTACT");
        removeExistingContact();
        System.out.println("UPDATING A NON-EXISTING CONTACT");
        updateNonExistingContact();
        System.out.println("REMOVING A NON-EXISTING CONTACT");
        removeNonExistingContact();
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            retrieveContact(contact.getName());
        }
    }

    public void createContacts() {
        for (Contact contact : contacts) {
            createContact(contact);
        }
    }

    public void createExistingContact() {
        createContact(contacts[0]);
    }

    public void updateExistingContact() {
        Contact contact = new Contact("Daniele","07890", "Mics Street 1");
        String message = "Contact " + contact.getName() + " updated!\n"
                        + contact;
        execute(()->fileManager.update(contact), message);
    }

    public void removeExistingContact() {
        String name = contacts[2].getName();
        String message = "Contact " + name + " removed!";
        execute(()->fileManager.remove(name), message);
    }
```

```java
    public void updateNonExistingContact() {
        Contact contact = new Contact("Bar", "07890", "Mics Street 1");
        String message = "Contact " + contact.getName() + " updated!";
        execute(()->fileManager.update(contact), message);
    }

    public void removeNonExistingContact() {
        String name = "Ligeia";
        String message = "Contact " + name + " removed!";
        execute(()->fileManager.remove(name), message);
    }

    public void retrieveNonExistingContact() {
        retrieveContact("Foo");
    }

    public void retrieveContact(String contactName) {
        execute(()->fileManager.retrieve(contactName));
    }

    private void createContact(Contact contact) {
        String message = "Contact " + contact.getName() + " created!\n"
                         + contact;
        execute(()->fileManager.insert(contact), message);
    }

    public <O> O execute(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.execute();
            System.out.println("Retrieved contact:\n" + output);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }

    public void execute(Executor executor, String message) {
        try {
            executor.execute();
            System.out.println(message);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public static void main(String args[]) {
        Exercise20K exercise20K = new Exercise20K();
        exercise20K.executeTest();
    }
}
```

Note that we had to create the two execute() methods of which, one takes a Retriever object, and the other an Executor object as input. These methods insert the call to the execute() method of Retriever or Executor within code block that manage exceptions that was previously replicated in each method. In this way, by passing a lambda expression containing the code to be executed to these two new methods, we were able to avoid duplications that were in the Exercise20J class.

The output follows:

```
TESTING THE CREATION OF THE THREE CONTACTS...
Daniele: contact already existing!
Giovanni: contact already existing!
Contact Ligeia created!
Name: Ligeia
Address:     Secrets Avenue 3
Phone:       07899921
RETRIEVING THE THREE CONTACTS...
Retrieved contact:
Name: Daniele
Address:     Mics Street 1
Phone:       07890
Retrieved contact:
Name: Giovanni
Address:     Sciences Square 2
Phone:       0565432190
Retrieved contact:
Name: Ligeia
Address:     Secrets Avenue 3
Phone:       07899921
TESTING THE CREATION OF AN ALREADY EXISTING CONTACT
Daniele: contact already existing!
TRYING TO RECOVER A NON-EXISTING CONTACT
Foo: contact not found!
UPDATING AN EXISTING CONTACT
Contact Daniele updated!
Name: Daniele
Address:     Mics Street 1
Phone:       07890
REMOVING AN EXISTING CONTACT
Contact Ligeia removed!
UPDATING A NON-EXISTING CONTACT
Bar: contact not found!
REMOVING A NON-EXISTING CONTACT
Ligeia: contact not found!
```

Solution 20.l)

A possible solution could be:

```
package phonebook.integration;
```

```java
import java.util.*;
import java.io.*;
import phonebook.exceptions.*;
import phonebook.util.*;
import phonebook.data.*;
import java.nio.file.*;

public class FileNIO2Manager implements SerializationManager<Contact> {

    @Override
    public void insert(Contact contact) throws DuplicateContactException,
                                        FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getFileName(contact.getName()));
        if (Files.exists(path)) {
            throw new DuplicateContactException(contact.getName() +
              ": contact already existing!");
        }
        register(contact);
    }

    @Override
    public Contact retrieve(String name) throws NonExistingContactException {
        Contact contact = getContact(FileUtils.getFileName(name));
        if (contact == null) {
            throw new NonExistingContactException(name + ": contact not found!");
        }
        return contact;
    }

    @Override
    public void update(Contact contact) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        if (isExistingContact(contact.getName())) {
            register(contact);
        } else {
            throw new NonExistingContactException(contact.getName() +
              ": contact not found!");
        }
    }

    @Override
    public void remove(String name) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getFileName(name));
        if (Files.exists(path)) {
            Files.delete(path);
            System.out.println("Contact " + name + " deleted!");
        } else {
            throw new NonExistingContactException(name + ": contact not found!");
        }
    }
```

660

```
    private void register(Contact contact) throws FileNotFoundException,
                                                    IOException {
        Path path = Paths.get(FileUtils.getFileName(contact.getName()));
        Files.write(path, getBytesFromObject(contact));
        System.out.println("Contact registered:\n" + contact);
    }

    private byte[] getBytesFromObject(Object object) throws IOException {
        try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
                ObjectOutput out = new ObjectOutputStream(bos)) {
            out.writeObject(object);
            return bos.toByteArray();
        }
    }

    private Object getObjectFromBytes(byte[] bytes) throws IOException,
                                                    ClassNotFoundException {
        try (ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
                ObjectInput in = new ObjectInputStream(bis)) {
            return in.readObject();
        }
    }

    private boolean isExistingContact(String name) {
        Path path = Paths.get(FileUtils.getFileName(name));
        return Files.exists(path);
    }

    private Contact getContact(String name) {
        Path path = Paths.get(name);
        byte[] bytes = null;
        Contact contact = null;
        try {
            bytes = Files.readAllBytes(path);
            contact = (Contact)getObjectFromBytes(bytes);
            System.out.println("Contact retrieved:\n" + contact);
        } catch (Exception exc) {
            return null;
        }
        return contact;
    }
}
```

The class for testing `Exercise20L`, differs from the class `Exercise20K` only by the application of the DIP principle which instantiates the new class `FileNIO2Manager`, using a reference of type `SerializationManager`. Everything else does not change:

```
package phonebook.test;
```

```
import phonebook.exceptions.*;
import phonebook.integration.*;
import phonebook.data.*;
import phonebook.util.*;

public class Exercise20L {

    private SerializationManager<Contact> fileManager;

    private Contact[] contacts;

    Exercise20L() {
        contacts = getContacts();
        fileManager = new FileNIO2Manager();
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS...");
        retrieveContacts();
        System.out.println("TESTING THE CREATION OF A ALREADY EXISTING CONTACT");
        createExistingContact();
        System.out.println("TRYING TO RECOVER A NON-EXISTING CONTACT");
        retrieveNonExistingContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateExistingContact();
        System.out.println("REMOVING AN EXISTING CONTACT");
        removeExistingContact();
        System.out.println("UPDATING A NON-EXISTING CONTACT");
        updateNonExistingContact();
        System.out.println("REMOVING A NON-EXISTING CONTACT");
        removeNonExistingContact();
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            retrieveContact(contact.getName());
        }
    }

    public void createContacts() {
        for (Contact contact : contacts) {
            createContact(contact);
        }
    }

    public void createExistingContact() {
        createContact(contacts[0]);
    }
```

```java
    public void updateExistingContact() {
        Contact contact = new Contact("Daniele","07890", "Mics Street 1");
        String message = "Contact " + contact.getName() + " updated!\n"
                            + contact;
        execute(()->fileManager.update(contact), message);
    }

    public void removeExistingContact() {
        String name = contacts[2].getName();
        String message = "Contact " + name + " removed!";
        execute(()->fileManager.remove(name), message);
    }

    public void updateNonExistingContact() {
        Contact contact = new Contact("Bar","07890", "Mics Street 1");
        String message = "Contact " + contact.getName() + " updated!";
        execute(()->fileManager.update(contact), message);
    }

    public void removeNonExistingContact() {
        String name = "Ligeia";
        String message = "Contact " + name + " removed!";
        execute(()->fileManager.remove(name), message);
    }

    public void retrieveNonExistingContact() {
        retrieveContact("Foo");
    }

    public void retrieveContact(String contactName) {
        execute(()->fileManager.retrieve(contactName));
    }

    private void createContact(Contact contact) {
        String message = "Contact " + contact.getName() + " created!\n"
                            + contact;
        execute(()->fileManager.insert(contact), message);
    }

    public <O> O execute(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.execute();
            System.out.println("Retrieved contact:\n" + output);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }
```

```
    public void execute(Executor executor, String message) {
        try {
            executor.execute();
            System.out.println(message);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public static void main(String args[]) {
        Exercise20L exercise20L = new Exercise20L();
        exercise20L.executeTest();
    }
}
```

We can return to the old version by simply replacing the instance of `FileNIO2Manager` with the instance of `FileManager`.

Solution 20.m)

We have implemented the `SerializationManagerFactory` class using the Reflection API. However, a solution that made use of a `switch` expression or any conditional construct would also have been valid:

```
package phonebook.integration;

import phonebook.data.Contact;
import java.util.Properties;
import java.io.*;

public class SerializationManagerFactory {
    private static final String PACKAGE = "phonebook.integration.";
    private static final String SERIALIZATION_METHOD = "phonebook.ser";
    private static Properties properties;
    static {
        properties = new Properties();
        try {
            loadProperties();
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static SerializationManager<Contact> getSerializationManager()
      throws InstantiationException, ClassNotFoundException,
            IllegalAccessException {
        String className = properties.getProperty(SERIALIZATION_METHOD);
```

```
        System.out.println("Loading the " + className + " class");
        Class<?> classObject = Class.forName(PACKAGE + className);
        return (SerializationManager<Contact>)classObject.newInstance();
    }

    private static void loadProperties() throws IOException {
        try (FileInputStream inputStream =
            new FileInputStream("config.properties");) {
            properties.load(inputStream);
        }
    }
}
```

Obviously, we assumed that the value of the property coincides with the name of the class to be loaded. For example, we could save the **config.properties** file with the following content:

```
phonebook.ser=FileManager
```

Finally we can test our implementation with the following class `Exercise20M` (as usual in bold there are changes from the version `Exercise20L`):

```
package phonebook.test;

import phonebook.exceptions.*;
import phonebook.integration.*;
import phonebook.data.*;
import phonebook.util.*;

public class Exercise20M {
    private SerializationManager<Contact> fileManager;

    private Contact[] contacts;

    Exercise20M() {
        contacts = getContacts();
        try {
            fileManager = SerializationManagerFactory.getSerializationManager();
        } catch (Exception exc) {
            exc.printStackTrace();
            System.exit(1);
        }
    }

    private void executeTest() {
        System.out.println("TESTING THE CREATION OF THE THREE CONTACTS...");
        createContacts();
        System.out.println("RETRIEVING THE THREE CONTACTS...");
        retrieveContacts();
        System.out.println("TESTING THE CREATION OF A ALREADY EXISTING CONTACT");
```

```java
        createExistingContact();
        System.out.println("TRYING TO RECOVER A NON-EXISTING CONTACT");
        retrieveNonExistingContact();
        System.out.println("UPDATING AN EXISTING CONTACT");
        updateExistingContact();
        System.out.println("REMOVING AN EXISTING CONTACT");
        removeExistingContact();
        System.out.println("UPDATING A NON-EXISTING CONTACT");
        updateNonExistingContact();
        System.out.println("REMOVING A NON-EXISTING CONTACT");
        removeNonExistingContact();
    }

    public void retrieveContacts() {
        for (Contact contact : contacts) {
            retrieveContact(contact.getName());
        }
    }

    public void createContacts() {
        for (Contact contact : contacts) {
            createContact(contact);
        }
    }

    public void createExistingContact() {
        createContact(contacts[0]);
    }

    public void updateExistingContact() {
        Contact contact = new Contact("Daniele","07890", "Mics Street 1");
        String message = "Contact " + contact.getName() + " updated!\n"
                            + contact;
        execute(()->fileManager.update(contact), message);
    }

    public void removeExistingContact() {
        String name = contacts[2].getName();
        String message = "Contact " + name + " removed!";
        execute(()->fileManager.remove(name), message);
    }

    public void updateNonExistingContact() {
        Contact contact = new Contact("Bar","07890", "Mics Street 1");
        String message = "Contact " + contact.getName() + " updated!";
        execute(()->fileManager.update(contact), message);
    }

    public void removeNonExistingContact() {
        String name = "Ligeia";
        String message = "Contact " + name + " removed!";
```

```
                execute(()->fileManager.remove(name), message);
    }

    public void retrieveNonExistingContact() {
        retrieveContact("Foo");
    }

    public void retrieveContact(String contactName) {
        execute(()->fileManager.retrieve(contactName));
    }

    private void createContact(Contact contact) {
        String message = "Contact " + contact.getName() + " created!\n"
                            + contact;
        execute(()->fileManager.insert(contact), message);
    }

    public <O> O execute(Retriever<O> retriever) {
        O output = null;
        try {
            output = retriever.execute();
            System.out.println("Retrieved contact:\n" + output);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
        return output;
    }

    public void execute(Executor executor, String message) {
        try {
            executor.execute();
            System.out.println(message);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public static void main(String args[]) {
        Exercise20M exercise20M = new Exercise20M();
        exercise20M.executeTest();
    }
}
```

Solution 20.n)

We decided to create the InvalidCommandException exception extending IOException in the following way:

```
package phonebook.exceptions;

import java.io.IOException;

public class InvalidCommandException extends IOException {

    private static final long serialVersionUID = 6742493040156525789L;

    public InvalidCommandException(String command) {
        super("Invalid command:" + command);
    }
}
```

Here it is the implementation of the PhonebookCLI class:

```
package phonebook.presentation;

import phonebook.data.*;
import phonebook.exceptions.*;
import java.util.*;
import java.io.*;

public class PhonebookCLI {
    private Scanner scanner;
    private static final String INSERT = "/i";
    private static final String SHOW_CONTACT = "/v";
    private static final String END = "/t";

    public PhonebookCLI() {
        scanner = new Scanner(System.in);
    }

    public void showContacts(List<Contact> contacts) {
        try {
            int numberOfContacts = contacts.size();
            if (numberOfContacts == 0) {
                System.out.println("No contact found");
            } else {
                System.out.println("Contacts list in the phonebook:");
                for (Contact contact : contacts) {
                    System.out.println(contact.getName());
                }
                System.out.printf("Type '%s' followed by the name of a contact,"
                                    + " to see the details\n", SHOW_CONTACT);
            }
            System.out.printf("Type '%s' to insert a new contact\n", INSERT);
            System.out.printf("Type '%s' to terminate the program\n", END);
            String command = scanner.nextLine();
```

```
            if (command.equals(INSERT)) {
                System.out.println("Insert contact method called");
            } else if (command.equals(END)) {
                System.out.println("Program terminated");
                System.exit(1);
            } else if (isValidCommandToShowAContact(contacts, command)) {
                System.out.printf("Show contact method called for %s\n", " +
                                    extractContactName(command));
            } else {
                throw new InvalidCommandException(command);
            }
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private boolean isValidCommandToShowAContact(List<Contact> contacts,
      String command) throws NonExistingContactException,
                            InvalidCommandException {
        boolean result = false;
        if (command.startsWith(SHOW_CONTACT)) {
            for (Contact contact : contacts) {
                if (command.endsWith(contact.getName())) {
                    return true;
                }
            }
        } else {
            throw new InvalidCommandException(command);
        }
        throw new NonExistingContactException(String.format(
          "Contact %s not found!", extractContactName(command)));
    }

    public String extractContactName(String command) {
        return command.substring(command.indexOf(" ")+1, command.length());
    }
}
```

Finally, this is the `Exercise20N` class:

```
package phonebook.test;

import phonebook.presentation.*;
import phonebook.data.*;
import phonebook.exceptions.*;
import java.util.*;

public class Exercise20N {
```

```
    private void executeTest() {
        PhonebookCLI cli = new PhonebookCLI();
        cli.showContacts(getContacts());
    }

    private List<Contact> getContacts() {
        List<Contact> contacts = new ArrayList<>();
        Contact contact1 = new Contact("Daniele", "01234560", "Guitars Street 1");
        Contact contact2 = new Contact("Giovanni", "0565432190", "Sciences Square 2");
        Contact contact3 = new Contact("Ligeia", "07899921", "Secrets Avenue 3");
        contacts.add(contact1);
        contacts.add(contact2);
        contacts.add(contact3);
        return contacts;
    }

    public static void main(String args[]) {
        Exercise20N exercise20N = new Exercise20N();
        exercise20N.executeTest();
    }
}
```

Solution 20.o)

First, let's start to modify the Model, adding the new getContacts() method to the SerializationManager interface:

```
package phonebook.integration;

import java.util.*;
import java.io.*;
import phonebook.exceptions.*;
import phonebook.util.*;
import phonebook.data.*;

public interface SerializationManager<T extends Data> {
    void insert(T data) throws IOException;

    T retrieve(String id) throws IOException, ClassNotFoundException;

    void update(T data) throws IOException;

    void remove(String id) throws IOException;

    List<Contact> getContacts() throws IOException;
}
```

Then we have to implement this method in its two implementations FileManager and

FileManagerNIO2. Here it is the new version of the FileManager class:

```java
package phonebook.integration;

import java.util.*;
import java.io.*;
import phonebook.exceptions.*;
import phonebook.util.*;
import phonebook.data.*;

public class FileManager implements SerializationManager<Contact> {
    @Override
    public void insert(Contact contact) throws DuplicateContactException,
                                        FileNotFoundException, IOException {
        Contact existingContact = getContact(contact.getName());
        if (existingContact != null) {
            throw new DuplicateContactException(contact.getName() +
                ": contact already existing!");
        }
        store(contact);
    }

    @Override
    public Contact retrieve(String name) throws NonExistingContactException {
        Contact contact = getContact(name);
        if (contact == null) {
            throw new NonExistingContactException(name + ": contact not found!");
        }
        return contact;
    }

    @Override
    public void update(Contact contact) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        if (isExistingContact(contact.getName())) {
            store(contact);
        } else {
            throw new NonExistingContactException(contact.getName() +
                ": contact not found!");
        }
    }

    @Override
    public void remove(String name) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        File file = new File(FileUtils.getFileName(name));
        if (!file.delete()) {
            throw new NonExistingContactException(name + ": contact not found!");
        }
    }
```

```
    @Override
    public List<Contact> getContacts() throws IOException {
        List<Contact> contacts = new ArrayList<Contact>();
        File directory = new File(".");
        for ( File file : directory.listFiles()) {
            if (file.isFile()) {
                String fileName = file.getName();
                if (fileName.endsWith(".con")) {
                    contacts.add(getContact(fileName));
                }
            }
        }
        return contacts;
    }

    private void store(Contact contact) throws  FileNotFoundException,
                                                IOException {
        try (FileOutputStream fos =
        new FileOutputStream (new File(FileUtils.getFileName(
        contact.getName()))));
        ObjectOutputStream s = new ObjectOutputStream (fos);) {
            s.writeObject (contact);
        }
    }

    private boolean isExistingContact(String name) {
        File file = new File(FileUtils.getFileName(name));
        return file.exists();
    }

    private Contact getContact(String name) {
        try (FileInputStream fis = new FileInputStream (new File(name));
        ObjectInputStream ois = new ObjectInputStream (fis);) {
            Contact contact = (Contact)ois.readObject();
            return contact;
        } catch (Exception exc) {
            exc.printStackTrace();
            return null;
        }
    }
}
```

While the following is the new version of the `GestoreFileNIO2` class:

```
package phonebook.integration;

import java.util.*;
import java.io.*;
import phonebook.exceptions.*;
```

```java
import phonebook.util.*;
import phonebook.data.*;
import java.nio.file.*;
import java.util.stream.*;

public class FileManagerNIO2 implements SerializationManager<Contact> {
    @Override
    public void insert(Contact contact) throws DuplicateContactException,
                                        FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getFileName(contact.getName()));
        if (Files.exists(path)) {
            throw new DuplicateContactException(contact.getName() +
                ": contact already existing!");
        }
        register(contact);
    }

    @Override
    public Contact retrieve(String name) throws NonExistingContactException {
        Contact contact = getContact(name);
        if (contact == null) {
            throw new NonExistingContactException(name + ": contact not found!");
        }
        return contact;
    }

    @Override
    public void update(Contact contact) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        if (isExistingContact(contact.getName())) {
            register(contact);
        } else {
            throw new NonExistingContactException(contact.getName() +
                ": contact not found!");
        }
    }

    @Override
    public void remove(String name) throws NonExistingContactException,
            DuplicateContactException, FileNotFoundException, IOException {
        Path path = Paths.get(FileUtils.getFileName(name));
        if (Files.exists(path)) {
            Files.delete(path);
        } else {
            throw new NonExistingContactException(name + ": contact not found!");
        }
    }
```

673

```java
    @Override
    public List<Contact> getContacts() throws IOException {
        List<Contact> contacts = new ArrayList<>();
        try (Stream<Path> walk = Files.walk(Paths.get("."))) {
            contacts = walk.map(p -> p.toString())
                .filter(f -> f.endsWith(".con")).map(
                f -> getContact(f)).collect(Collectors.toList());
        }
        return contacts;
    }

    private void register(Contact contact) throws  FileNotFoundException,
                                                    IOException {
        Path path = Paths.get(FileUtils.getFileName(contact.getName()));
        Files.write(path, getBytesFromObject(contact));
    }

    private byte[] getBytesFromObject(Object object) throws IOException {
        try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutput out = new ObjectOutputStream(bos)) {
            out.writeObject(object);
            return bos.toByteArray();
        }
    }

    private Object getObjectFromBytes(byte[] bytes) throws IOException,
                                                    ClassNotFoundException {
        try (ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
        ObjectInput in = new ObjectInputStream(bis)) {
            return in.readObject();
        }
    }

    private boolean isExistingContact(String name) {
        Path path = Paths.get(FileUtils.getFileName(name));
        return Files.exists(path);
    }

    private Contact getContact(String name) {
        Path path = Paths.get(FileUtils.getFileName(name));
        byte[] bytes = null;
        Contact contact = null;
        try {
            bytes = Files.readAllBytes(path);
            contact = (Contact)getObjectFromBytes(bytes);
        }
        catch (Exception exc) {
            return null;
        }
```

```
            return contact;
        }
    }
```

The new version of the PhonebookCLI class follows, which contains the PhonebookController nested class:

```
package phonebook.presentation;

import phonebook.data.*;
import phonebook.exceptions.*;
import phonebook.integration.*;
import java.util.*;
import java.io.*;

public class PhonebookCLI {
    private Scanner scanner;
    private PhonebookController controller;
    private static final String INSERT = "/i";
    private static final String SHOW_CONTACT = "/v";
    private static final String END = "/t";

    public PhonebookCLI() {
        scanner = new Scanner(System.in);
        controller = new PhonebookController();
    }

    public void start() {
        try {
            controller.start();
        } catch (IOException exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void showContacts(List<Contact> contacts) {
        try {
            int numberOfContacts = contacts.size();
            if (numberOfContacts == 0) {
                System.out.println("No contact found");
            } else {
                System.out.println("Contacts list in the phonebook:");
                for (Contact contact : contacts) {
                    System.out.println(contact.getName());
                }
                System.out.printf("Type '%s' followed by the name of a contact,"
                  + " to see the details\n", SHOW_CONTACT);
            }
            System.out.printf("Type '%s' to insert a new contact\n", INSERT);
```

```
            System.out.printf("Type '%s' to terminate the program\n", END);
            String command = scanner.nextLine();
            if (command.equals(INSERT)) {
                System.out.println("Insert contact method called");
            } else if (command.equals(END)) {
                System.out.println("Program terminated");
                System.exit(1);
            } else if (isValidCommandToShowAContact(contacts, command)) {
                System.out.printf("Show contact method called for %s\n", +
                    "extractContactName(command));
            } else {
                throw new InvalidCommandException(command);
            }
        } catch (Exception exc) {
            exc.printStackTrace();
            System.out.println(exc.getMessage());
        }
    }

    private boolean isValidCommandToShowAContact(List<Contact> contacts,
        String command) throws NonExistingContactException,
                            InvalidCommandException {
        boolean result = false;
        if (command.startsWith(SHOW_CONTACT)) {
            for (Contact contact : contacts) {
                if (command.endsWith(contact.getName())) {
                    return true;
                }
            }
        } else {
            throw new InvalidCommandException(command);
        }
        throw new NonExistingContactException(String.format(
            "Contact %s not found!", extractContactName(command)));
    }

    public String extractContactName(String command) {
        return command.substring(command.indexOf(" ")+1, command.length());
    }

    class PhonebookController {
        SerializationManager fileManager;
        public PhonebookController () {
            try {
                fileManager =
                    SerializationManagerFactory.getSerializationManager();
            } catch (Exception exc) {
                exc.printStackTrace();
                System.exit(1);
```

```
                }
            }

            public void start() throws IOException {
                List<Contact> contacts = fileManager.getContacts();
                showContacts(contacts);
            }
        }
    }
```

Soluzione 20.p)

First, we have defined the EmptyNameException exception requested like this:

```
package phonebook.exceptions;

public class EmptyNameException extends Exception {
    private static final long serialVersionUID = 8290498479156525745L;

    public EmptyNameException() {
        super("The contact name cannot be empty!");
    }
}
```

Obviously, we have implemented the check of the empty name, within the Contact class using encapsulation:

```
package phonebook.data;

import phonebook.exceptions.*;

public class Contact extends Entity {
    protected static final String UNKNOWN = "unknown";
    private String name;
    private String phoneNumber;
    private String address;

    public Contact(String name, String phoneNumber) throws EmptyNameException {
        this.setName(name);
        this.setPhoneNumber(phoneNumber);
        this.setAddress(UNKNOWN);
    }

    public Contact(String name, String phoneNumber, String address)
      throws EmptyNameException {
        this.setName(name);
        this.setPhoneNumber(phoneNumber);
        this.setAddress(address);
    }
```

677

```
    public void setAddress(String address) {
        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setName(String name) throws EmptyNameException {
        if (name == null || name.trim().length() == 0) {
            throw new EmptyNameException();
        }
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Name:\t" + name + "\nAddress:\t" + address + "\nPhone:\t" +
          phoneNumber;
    }
}
```

As for the changes of the PhonebookCLI and PhonebookController classes required, we highlight them below in bold:

```
package phonebook.presentation;

import phonebook.data.*;
import phonebook.exceptions.*;
import phonebook.integration.*;
import java.util.*;
import java.io.*;

public class PhonebookCLI {
    private Scanner scanner;
    private PhonebookController controller;
    private static final String INSERT = "/i";
```

```java
    private static final String SHOW_CONTACT = "/v";
    private static final String EXECUTE = "/e";
    private static final String BACK = "/b";
    private static final String END = "/t";

    public PhonebookCLI() {
        scanner = new Scanner(System.in);
        controller = new PhonebookController();
    }

    public void start() {
        try {
            controller.start();
        } catch (IOException exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void showContacts(List<Contact> contacts) {
        try {
            int numberOfContacts = contacts.size();
            if (numberOfContacts == 0) {
                System.out.println("No contact found");
            } else {
                System.out.println("Contacts list in the phonebook:");
                int size = contacts.size();
                for (int i = 1; i <= size; ++i) {
                    System.out.println(i + "\t" +contacts.get(i-1).getName());
                }
                System.out.printf("Type '%s' followed by the name of a contact,"
                    + " to see the details\n", SHOW_CONTACT);
            }
            System.out.printf("Type '%s' to insert a new contact\n", INSERT);
            System.out.printf("Type '%s' to terminate the program\n", END);
            String command = scanner.nextLine();
            controller.handleShowContacts(command, contacts);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void insertContact() {
        try {
            String name = getContactData("name");
            String telephoneNumber = getContactData("telephone number");
            String address = getContactData("address");
            System.out.printf("Type '%s' to confirm the insertion\n", EXECUTE);
            System.out.printf("Type '%s' to go back and show contacts\n", BACK);
            String command = scanner.nextLine();
```

679

```
                controller.handleInsertContact(command, name, telephoneNumber,
                                                address);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void showConfirmation(String message) {
        try {
            System.out.println("Operation Confirmed!");
            showMessage(message);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    private void showMessage(String message) throws IOException,
                                                    InvalidCommandException {
        System.out.println(message + "\n");
        System.out.printf("Type '%s' to go back and show contacts\n", BACK);
        System.out.printf("Type '%s' to terminate the program\n", END);
        String command = scanner.nextLine();
        controller.handleShowConfirmation(command);
    }

    private String getContactData(String data) {
        String message = String.format("Insert the %s of the contact:", data);
        return getUserInput(message);
    }

    private String getUserInput(String message) {
        System.out.printf(message);
        return scanner.nextLine();
    }

class PhonebookController {
    SerializationManager fileManager;
    public PhonebookController () {
        try {
            fileManager =
                SerializationManagerFactory.getSerializationManager();
        } catch (Exception exc) {
            exc.printStackTrace();
            System.exit(1);
        }
    }

    public void start() throws IOException {
        List<Contact> contacts = fileManager.getContacts();
```

```
        showContacts(contacts);
    }

    public void handleShowContacts(String command, List<Contact> contacts)
                                                    throws IOException {
        if (command.equals(INSERT)) {
            insertContact();
        } else if (command.equals(END)) {
            System.out.println("Program terminated");
            System.exit(1);
        } else if (isValidCommandToShowAContact(contacts, command)) {
            System.out.printf("Show contact method called for %s\n",
                extractContactName(command));
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleInsertContact(String command, String name,
        String telephoneNumber, String address) throws IOException,
                                                    EmptyNameException {
        if (command.equals(EXECUTE)) {
            Contact contact = new Contact(name, telephoneNumber, address);
            fileManager.insert(contact);
            showConfirmation("Contact inserted:\n" + contact);
        } else if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleShowConfirmation(String command) throws IOException,
                                                    InvalidCommandException {
        if (command.equals(END)) {
            System.out.println("Program terminated");
            System.exit(1);
        } else if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
        } else {
            throw new InvalidCommandException(command);
        }
    }

    private boolean isValidCommandToShowAContact(List<Contact> contacts,
        String command) throws NonExistingContactException,
                                InvalidCommandException {
        boolean result = false;
        if (command.startsWith(SHOW_CONTACT)) {
```

```
                    for (Contact contact : contacts) {
                        if (command.endsWith(contact.getName())) {
                            return true;
                        }
                    }
                } else {
                    throw new InvalidCommandException(command);
                }
                throw new NonExistingContactException(String.format(
                  "Contact %s not" + " found!", extractContactName(command)));
            }

            public String extractContactName(String command) {
                return command.substring(command.indexOf(" ")+1, command.length());
            }
        }
    }
```

Finally, follows the Exercise20P:

```
package phonebook.test;

import phonebook.presentation.*;
import phonebook.data.*;
import phonebook.exceptions.*;
import java.util.*;

public class Exercise20P {

    private void executeTest() {
        PhonebookCLI cli = new PhonebookCLI();
        cli.start();
    }

    public static void main(String args[]) {
        Exercise20P exercise20P = new Exercise20P();
        exercise20P.executeTest();
    }
}
```

Solution 20.q)

The PhonebookCLI and PhonebookController classes have been modified as follows (changes in bold):

```
package phonebook.presentation;

import phonebook.data.*;
```

```java
import phonebook.exceptions.*;
import phonebook.integration.*;
import java.util.*;
import java.io.*;

public class PhonebookCLI {
    private Scanner scanner;
    private PhonebookController controller;
    private static final String INSERT = "/i";
    private static final String SHOW_CONTACT = "/v";
    private static final String UPDATE_CONTACT = "/u";
    private static final String REMOVE_CONTACT = "/r";
    private static final String EXECUTE = "/e";
    private static final String BACK = "/b";
    private static final String END = "/t";

    public PhonebookCLI() {
        scanner = new Scanner(System.in);
        controller = new PhonebookController();
    }

    public void start() {
        try {
            controller.start();
        } catch (IOException exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void showContact(Contact contact) {
        try {
            System.out.println("Details of the contact selected:\n" + contact);
            System.out.printf("Type '%s' to update the contact\n",
                UPDATE_CONTACT);
            System.out.printf("Type '%s' to remove the contact\n",
                REMOVE_CONTACT);
            System.out.printf("Type '%s' to go back and show contacts\n", BACK);
            String command = scanner.nextLine();
            controller.handleShowContact(command, contact);
        } catch (Exception exc) {
            showError(exc);
        }
    }

    public void updateContact(Contact contact) {
        try {
            String telephoneNumber = getContactData("telephone number");
            String address = getContactData("address");
            System.out.printf("Type '%s' to confirm the update\n", EXECUTE);
```

683

```java
            System.out.printf("Type '%s' to go back and show contacts\n", BACK);
            String command = scanner.nextLine();
            controller.handleUpdateContact(command, contact.getName(),
              telephoneNumber, address);
        } catch (Exception exc) {
            showError(exc);
        }
    }

    public void removeContact(Contact contact) {
        try {
            System.out.printf("Type '%s' to confirm the removal\n", EXECUTE);
            System.out.printf("Type '%s' to go back and show contacts\n",
              BACK);
            String command = scanner.nextLine();
            controller.handleRemoveContact(command, contact.getName());
        }
        catch (Exception exc) {
            showError(exc);
        }
    }

    public void showContacts(List<Contact> contacts) {
        try {
            int numberOfContacts = contacts.size();
            if (numberOfContacts == 0) {
                System.out.println("No contact found");
            } else {
                System.out.println("Contacts list in the phonebook:");
                int size = contacts.size();
                for (int i = 1; i <= size; ++i) {
                    System.out.println(i + "\t" + contacts.get(i-1).getName());
                }
                System.out.printf("Type '%s' followed by the name of a contact,"
                  + " to see the details\n", SHOW_CONTACT);
            }
            System.out.printf("Type '%s' to insert a new contact\n", INSERT);
            System.out.printf("Type '%s' to terminate the program\n", END);
            String command = scanner.nextLine();
            controller.handleShowContacts(command, contacts);
        } catch (Exception exc) {
            System.out.println(exc.getMessage());
        }
    }

    public void insertContact() {
        try {
            String name = getContactData("name");
            String telephoneNumber = getContactData("telephone number");
```

```
        String address = getContactData("address");
        System.out.printf("Type '%s' to confirm the insertion\n", EXECUTE);
        System.out.printf("Type '%s' to go back and show contacts\n", BACK);
        String command = scanner.nextLine();
        controller.handleInsertContact(command, name, telephoneNumber,
            address);
    } catch (Exception exc) {
        System.out.println(exc.getMessage());
    }
}

public void showConfirmation(String message) {
    System.out.println("Operation confirmed!");
    showMessage(message);
}

public void showError(Exception exc) {
    System.out.println("There's a problem!");
    showMessage(exc.toString());
}

private void showMessage(String message) {
    try {
        System.out.println(message + "\n");
        System.out.printf("Type '%s' to go back and show contacts\n",
            BACK);
        System.out.printf("Type '%s' to terminate the program\n", END);
        String command = scanner.nextLine();
        controller.handleShowMessage(command);
    }
    catch (Exception exc) {
        showMessage(message);
    }
}

private String getContactData(String data) {
    String message = String.format("Insert the %s of the contact:", data);
    return getUserInput(message);
}

private String getUserInput(String message) {
    System.out.printf(message);
    return scanner.nextLine();
}

class PhonebookController {
    SerializationManager<Contact> fileManager;
    public PhonebookController () {
        try {
```

685

```
            fileManager =
                SerializationManagerFactory.getSerializationManager();
        } catch (Exception exc) {
            exc.printStackTrace();
            System.exit(1);
        }
    }

    public void start() throws IOException {
        List<Contact> contacts = fileManager.getContacts();
        showContacts(contacts);
    }

    public void handleShowContacts(String command, List<Contact> contacts)
      throws IOException, ClassNotFoundException {
        if (command.equals(INSERT)) {
            insertContact();
        } else if (command.equals(END)) {
            System.out.println("Program terminated");
            System.exit(1);
        } else if (isValidCommandToShowAContact(contacts, command)) {
            String contactName = extractContactName(command);
            System.out.printf("Show contact method called for %s\n",
              contactName);
            Contact contact = fileManager.retrieve(contactName);
            showContact(contact);
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleShowContact(String command,Contact contact)
      throws IOException, ClassNotFoundException {
        if (command.equals(UPDATE_CONTACT)) {
            updateContact(contact);
        } else if (command.equals(REMOVE_CONTACT)) {
            removeContact(contact);
        } else if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleInsertContact(String command, String name, String
      telephoneNumber, String address) throws IOException, EmptyNameException {
        if (command.equals(EXECUTE)) {
            Contact contact = new Contact(name, telephoneNumber, address);
            fileManager.insert(contact);
```

```
            showConfirmation("Contact inserted:\n" + contact);
        } else if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleUpdateContact(String command, String name,
       String telephoneNumber, String address) throws IOException,
                                                  EmptyNameException {
        if (command.equals(EXECUTE)) {
            Contact contact = new Contact(name, telephoneNumber, address);
            fileManager.update(contact);
            showConfirmation("Update confirmed:\n" + contact);
        } else if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleRemoveContact(String command, String name)
                                                  throws IOException {
        if (command.equals(EXECUTE)) {
            fileManager.remove(name);
            showConfirmation("Removal confirmed:\n" + name);
        } else if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleShowConfirmation(String command) throws IOException,
                                                  InvalidCommandException {
        if (command.equals(END)) {
            System.out.println("Program terminated");
            System.exit(1);
        } else if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
        } else {
            throw new InvalidCommandException(command);
        }
    }

    public void handleShowMessage(String command) throws IOException {
        if (command.equals(BACK)) {
            showContacts(fileManager.getContacts());
```

687

```
            } else if (command.equals(BACK)) {
                showContacts(fileManager.getContacts());
            } else {
                throw new InvalidCommandException(command);
            }
        }

        private boolean isValidCommandToShowAContact(List<Contact> contacts,
          String command) throws NonExistingContactException,
                                    InvalidCommandException {
            boolean result = false;
            if (command.startsWith(SHOW_CONTACT)) {
                for (Contact contact : contacts) {
                    if (command.endsWith(contact.getName())) {
                        return true;
                    }
                }
            } else {
                throw new InvalidCommandException(command);
            }
            throw new NonExistingContactException(
              String.format("Contact %s not" + " found!",
              extractContactName(command)));
        }

        public String extractContactName(String command) {
            return command.substring(command.indexOf(" ")+1, command.length());
        }
    }
}
```

Obviously, the Exercise20Q class is the following:

```
package phonebook.test;

import phonebook.presentation.*;
import phonebook.data.*;
import phonebook.exceptions.*;
import java.util.*;

public class Exercise20Q {

    private void executeTest() {
        PhonebookCLI cli = new PhonebookCLI();
        cli.start();
    }

    public static void main(String args[]) {
        Exercise20Q exercise20Q = new Exercise20Q();
```

```
            exercise20Q.executeTest();
     }
 }
```

Solution 20.r)

The correct statements are the first and fourth. Statement 2 is incorrect since the getParent() returns a string that represents a directory. Statement 3 is incorrect because pathSeparator is a static variable of the File class, which represents the path separator dependent on the operating system. It could be deduced both from the name which is written in lower case (the constants instead are conventionally written in upper case), and from the fact that it cannot assume an a priori value since it depends on the operating system.

Solution 20.s)

The only correct statement is the first. Statement 2 is incorrect because Reader and Writer are interfaces and not classes. Statement 3 is incorrect because the readLine() method is defined within the BufferedReader class. Statement number 4 is incorrect since the constructor of an ObjectInputStream can take as input any object of type InputStream.

Solution 20.t)

The only incorrect statement is number 1. Statement number 2 is correct even if the efficiency and convenience of reading would undoubtedly be improved by "connecting" a BufferedReader to the FileReader. For example we could read a file with a code similar to the following:

```
Reader fileReader = new FileReader("c:/myfile.txt");
int data = fileReader.read();
while (data != -1) {
    System.out.print(data);
    data = fileReader.read();
}
fileReader.close();
```

Statement number 3 is correct, because the File class defines the mkDir() method, and also the 4 is correct because the File class defines the list() method.

Solution 20.u)

The correct answer is number 2, which is 12 bytes. In fact, an int type (32 bit = 4 byte) and a double (64 bit = 8 byte) are written within the file.

Solution 20.v)

The solution could be the following:

```
import java.io.*;

public class Solution20S {
    public static void main(String args[]) throws Exception {
        try (FileInputStream fis = new FileInputStream("new file.txt");
            DataInputStream dis = new DataInputStream(fis);) {
            for (int i = 0; i < 50; i++) {
                System.out.print(dis.readInt());
            }
        }
    }
}
```

The output will be:

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
20
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
```

```
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
```

Solution 20.w)

The correct statements are numbers 2 and 3. The 1 is incorrect because there are `Reader` implementations, such as `StringReader`, that do not declare throws clauses to `IOException`. Statement 4 is incorrect as only `Writer` implementations dealing with files such as `FileWriter` need to manage these implementations.

Solution 20.x)

The correct statements are numbers 1, 3 and 4. Statement 2 is incorrect because we have seen, also in paragraph 20.4.1, that it is possible to obtain the same result by linking a `BufferedReader` to an `InputStreamReader` whose reading source is the `System.in` object (i.e. the default input device, probably the keyboard). For convenience, the class defined in paragraph 20.4.1 `KeyboardInput` is shown below:

```java
import java.io.*;

public class KeyboardInput {
    public static void main (String args[]) throws IOException{
        String string = null;
        System.out.println("Type something then press enter...\nType \"end\" to"
                                            + " terminate the program.");
        try (InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader in = new BufferedReader(ir)) {
            string = in.readLine();
            while ( string != null ) {
                if (string.equals("end")) {
                    System.out.println("Program terminated");
                    break;
                }
```

```
                        System.out.println("You wrote: " + string);
                        string = in.readLine();
                    }
                }
            }
        }
```

Solution 20.y)

Only statements 1 and 3 are incorrect. In particular, statement 2 is correct since the existence of the getParent() method. The 3 instead is incorrect since the getOwner() method is defined by the Files class.

Solution 20.z)

Only statements 2 and 5 are correct. Statement 1 is incorrect because the Files list() method returns a Stream of Path objects. The 3 is incorrect because the Files and Path types belong to the java.nio.file package. While 4 is incorrect because Files is an abstract class.

Solution 20.aa) Networking, True or False:

1. **True**.

2. **True**.

3. **True**.

4. **True**.

5. **True**.

Chapter 21
Exercises

Java Database Connectivity

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Chapter 21
Exercise Solutions

Java Database Connectivity

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Chapter 22
Exercises

Java and the XML World

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Chapter 22
Exercise Solutions

Java and the XML World

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Chapter 23
Exercises

Graphical User Interfaces (GUIs)

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Chapter 23
Exercise Solutions

Graphical User Interfaces (GUIs)

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Chapter 24
Exercises

Introduction to JavaFX

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Chapter 24
Exercise Solutions

Introduction to JavaFX

To create and fix the exercises of this chapter faster, we have decided to publish them in html format on **https://www.javaforaliens.com**. Please visit the website to find the new exercises. If you subscribe to the telegram channel of Java for Aliens (**https://t.me/java4aliens**), you will be notified of all updates.

Appendix E
Exercises

The CLASSPATH Environment Variable

Package the authentication package designed and built in exercise 5.z.

Appendix E
Exercise Solutions

The CLASSPATH Environment Variable

The command to package the files is:

```
jar -cvfm authentication.jar manifest.txt com/claudiodesio/authentication/*.class
```

where **manifest.txt** is a file (which must be present in the folder from where the command is being executed):

```
Main-Class: com.claudiodesio.authentication.Authentication
```

> **Note that you have to wrap after the line, otherwise the command will not be executed.**

To run the application, use the following command:

```
java -jar authentication.jar
```

711