28 Terminal Programs

28.1 Introduction

Terminal programs seem to be quite simple: They do not have a graphical user interface and are therefore ideally suited as an introduction to Java programming. But even in the professional field there is always a need for programs for which the expense of a graphical interface is not worthwhile.

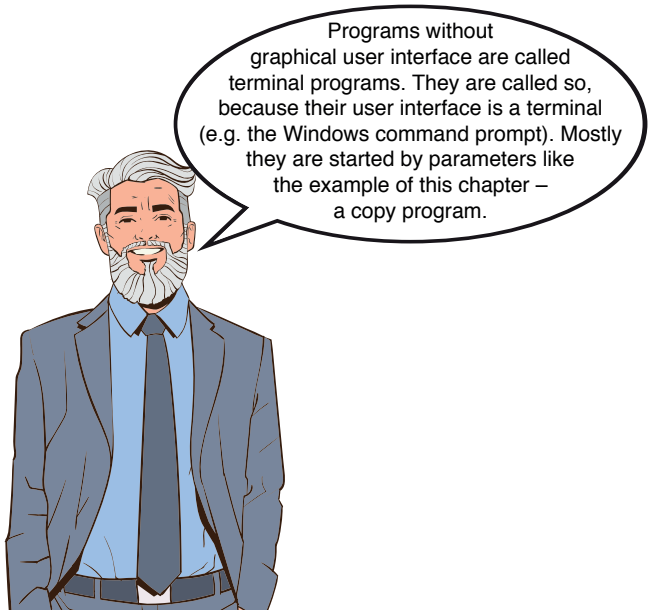

Figure 28.1 Professor Roth is a fan of terminal programs.

28.2 Requirements

The *Transfer* program is to copy part of a file tree from one directory to another (Figure 28.2). The program should be able to transfer the directory tree recursively. Recursive means that the program should also copy all subdirectories with the complete contents. Source and target directory should be passed via command line parameters. The program must perform a minimal error check so that nonsensical entries do not lead to loss of data.

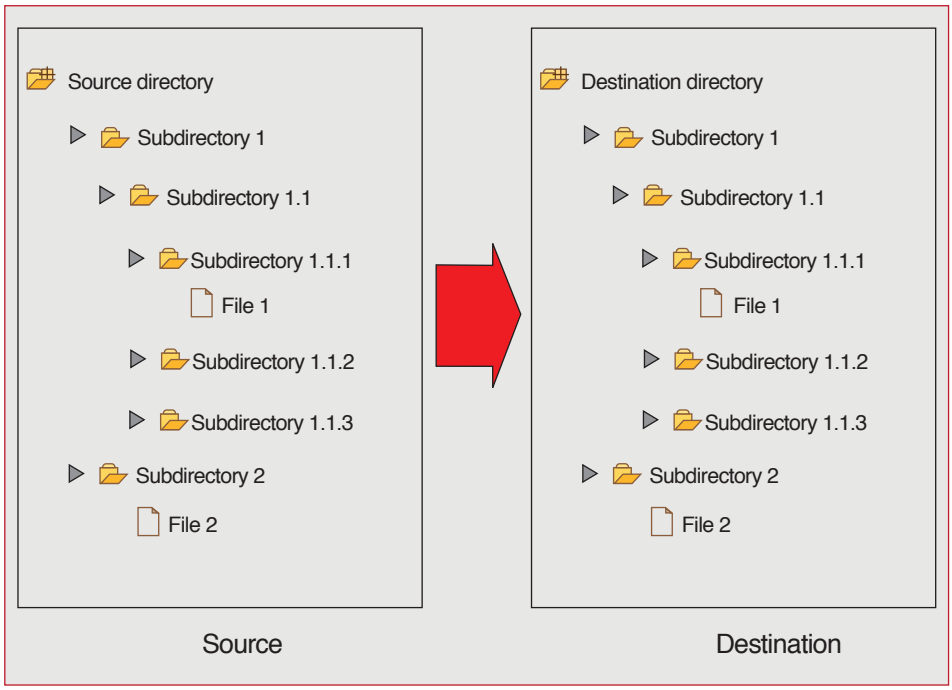

Figure 28.2 The requirement is that the program should copy complete directory trees.

If the user has not entered any parameters, the program should search for a configuration file in which the copy parameters are stored. The program shall create a backup copy of the work done during the day. It is therefore not started manually, but when the user wants to shut down his operating system (system shutdown). Summa summarum: Transfer is a program that creates a backup copy of a complete directory tree at the end of a working day.

28.3 Analysis and Design

The program is divided into three technically different areas: reading in the configuration, the copy logic and the event control. Let's start with reading the configuration.

28.3.1 Reading the Configuration

A Java program like this copy program must be built in such a way that the source and target directory is not hard coded in the program. Instead, you have to build the program in such a way that it reads in the source and target directories in a suitable way. For example, the configuration can be injected into the program via the command line using parameters. Another possibility is to read in a configuration file. The copy program should realize both. There are three main cases, which are shown in the process diagram in Figure 28.3 .

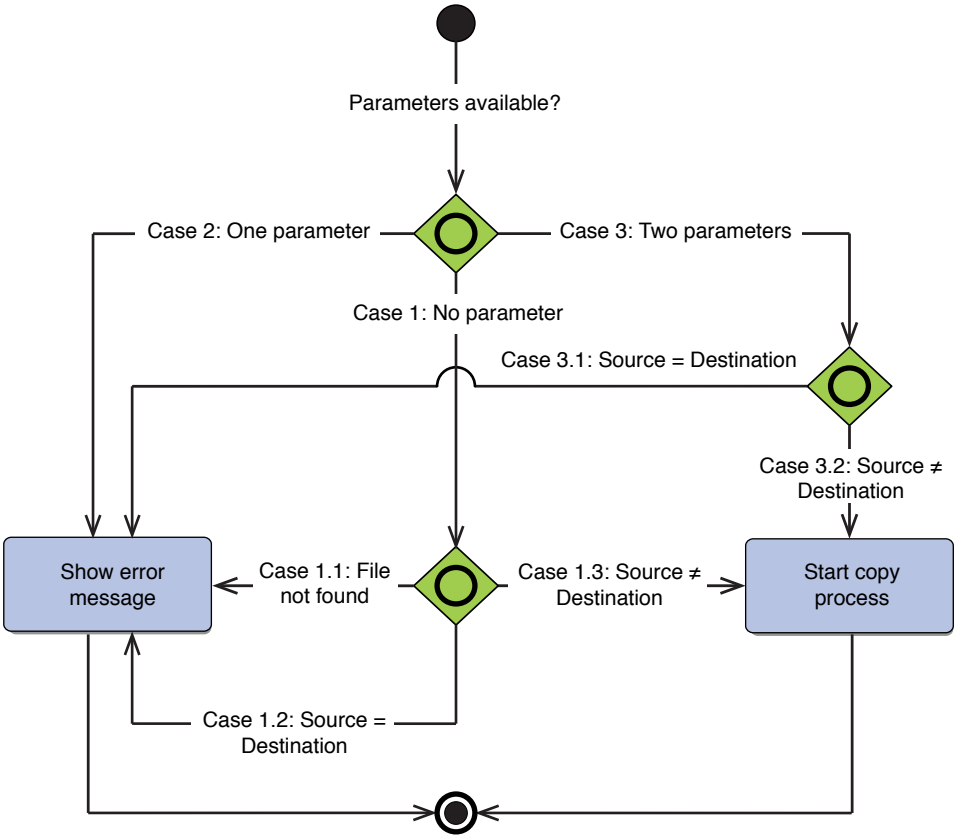

Figure 28.3 The process diagram of the overall flow.

The process diagram distinguishes three main cases: In *case 1* in the middle of the diagram, the user has not entered any parameters. This is either because he does not know the program and has simply started without parameters. However, this can also be due to the fact that he wants to read in the start parameters via a configuration file. In this case the program tries to find the configuration file and to evaluate the copy parameters stored in it. If the file cannot be found or is destroyed (*Case 1.1*), the program issues an error message and says goodbye. If the parameters are set so that the source is equal to the destination (*Case 1.2*), the program also issues an error message and exits. Only if the source is not equal to the destination, Transfer starts the copy process.

Case 2 is easier to handle: Here, the user has entered only one parameter. The program interprets this as an operating error, issues a corresponding message and says goodbye. In *Case 3*, the user has entered two parameters. If the source is the same as the destination (*Case 3.1*), the program issues an error message and also exits. However, if source and destination are different (case 3.2), the program starts the copying process.

28.3.2 Copy Logic

For the solution of the copy problem it is necessary to use file streams introduced by chapter 22, »Class Libraries«. To avoid the need for intermediate storage, the program should first read the source directory and then immediately write the destination directory. The entire copying process should run in a recursive algorithm.

28.4 Implementation

28.4.1 Main Class »TransferApp«

Start Eclipse With a New Workspace

Now please launch Eclipse and select the workspace named *Terminal_Programs* when the ECLIPSE IDE LAUNCHER appears.

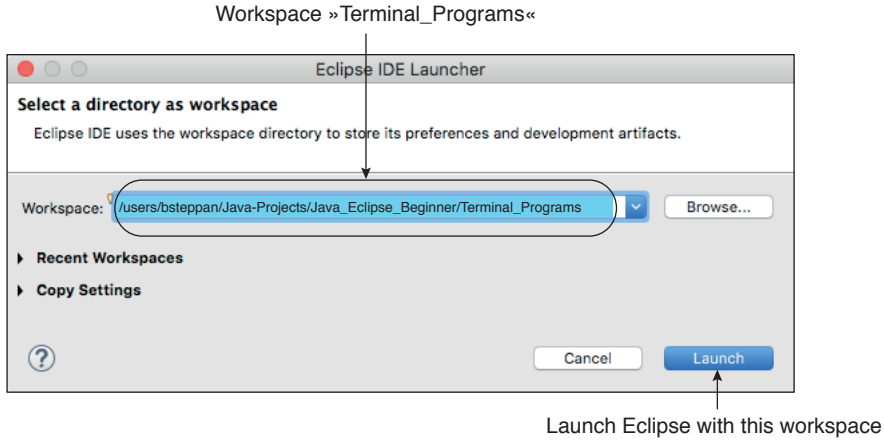

Figure 28.4 Select a new workspace.

Open the Dialog »New Java Project«

Then open the wizard for creating a new Java project (Figure 28.5). In this dialog, enter the title »Transfer« as the project name. Then check whether the dialog shows at least Java 13 as the Java Runtime Environment (JRE).

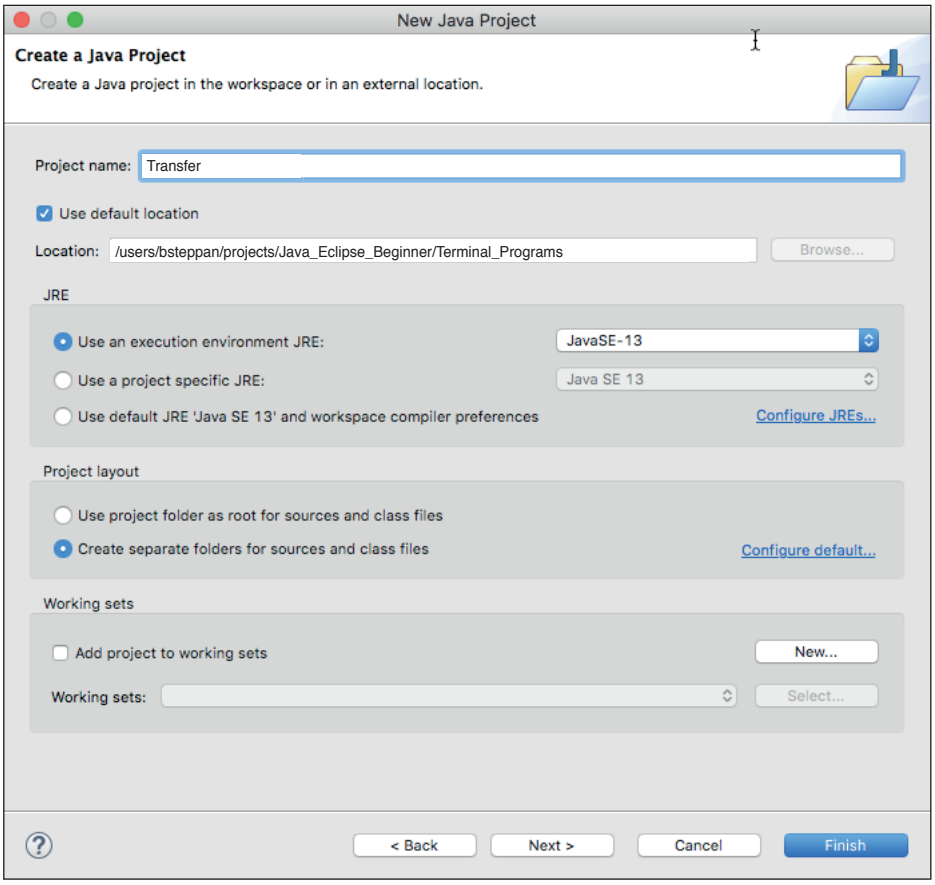

Figure 28.5 Create new Java project.

Deselect Modules

On the second page of the dialog you have to deselect the option CREATE MODULE-INFO.JAVA FILE. It is not needed for this example. The remaining settings of the dialog can be left as they are displayed in the development environment. Afterwards please create the project by clicking on FINISH.

Call the Dialog »New Java Class«

Now please expand the new project in the PACKAGE EXPLORER on the left side of Eclipse. Inside the project you will find the node SRC. Since you have not yet created a class the directory is still empty, of course. To change this, right-click on the directory. Then select the menu command NEW → CLASS (Figure 28.6).

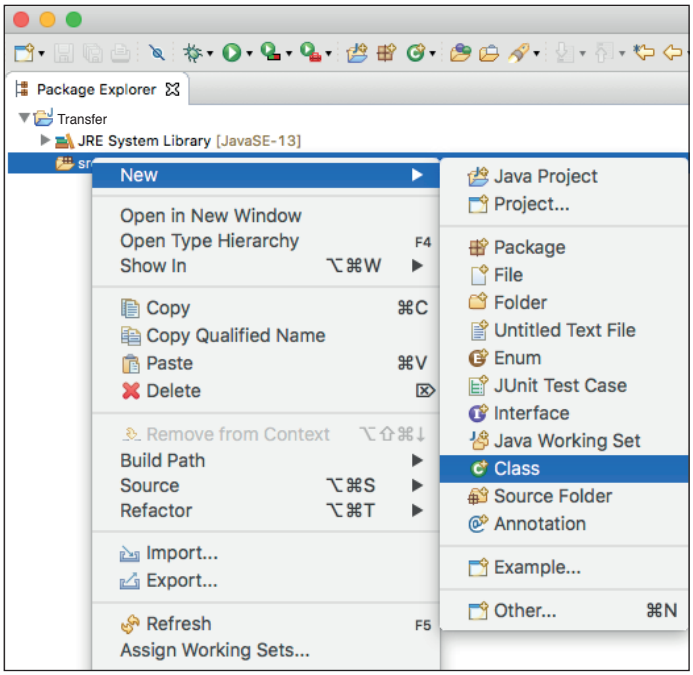

Figure 28.6 The menu command for calling the »New Java Class« dialog.

Package Structure

The class *TransferApp* belongs to the package *programmingcourse*. For a project like Transfer, which consists of only two classes, further subdivision is unnecessary. Also superfluous are object or class variables.

Used Classes

Only three direct class imports from the pool of the Java class libraries are needed. These used classes are used to read in the property file. If this fails, a *IOException* is thrown. It will be caught by the program, of course.

Create New Class »TransferApp«

The dialog for creating a class contains a field in the upper area called SOURCE FOLDER. This is the directory of the source code where Eclipse will store the new class. The second field denotes the name of the package to which the new class should belong. Please choose here the package name *programmingcourse*. The next field ENCLOSING TYPE is not important for this exercise and can be left empty. In the next field you have to enter the name of the class *TransferApp*. Use the radio buttons named MODIFIERS to set the access level of the new class. You can leave the default value *public* as it is here. The default *public* value makes the class accessible outside the package from any other class.

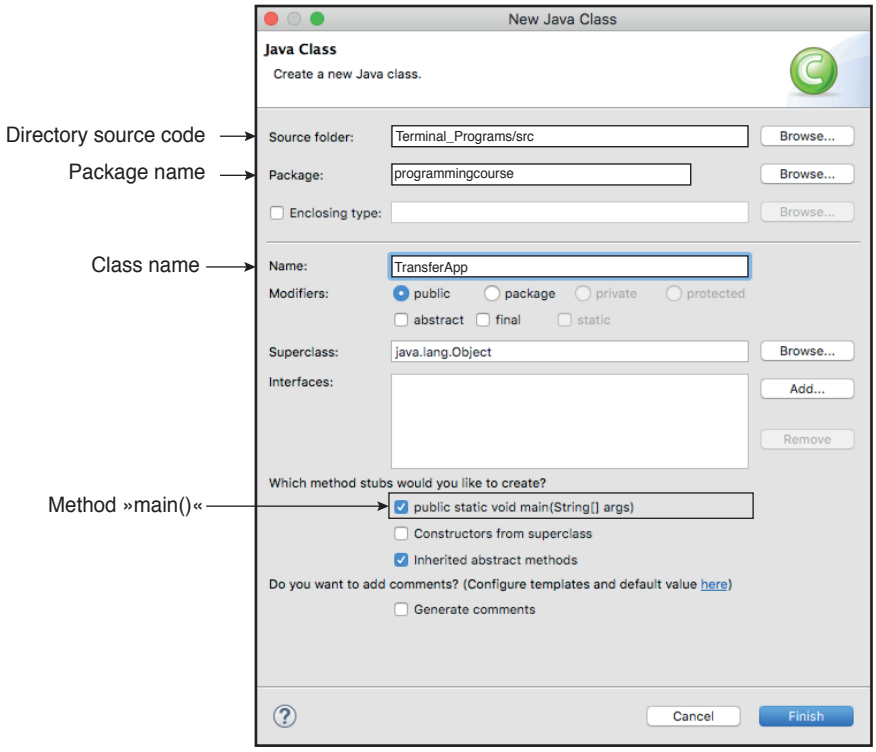

Figure 28.7 Create the new class »TransferApp« with a method »main()«.

Finally, select that Eclipse creates the method *main()*. If you have set this option, please click FINISH now to close the dialog and create the class. The class is of course still empty. It should first get a constructor.

Constructor

The constructor of the *TransferApp* class consists of two parts: the call to the copy method of the class *CopyLogic* and the exception handling if this call fails. The exception handling catches an error of the type IOException, which can occur whenever an operation for input or output fails.

Listing 28.1 The constructor creates the transfer program.

```
16   public TransferApp(String source, String destination) {
17     try {
18       CopyLogic.copy(source, destination);
19     } catch (IOException exception) {
20       System.out.println("Error while copying (directories correct?)"
21         + ": " + exception.getMessage());
22       exception.printStackTrace();
23     }
24   }
```

Method »showHelp()«

The method *showHelp()* is used to help the user by providing information. It is called whenever there is an error input by the user or within the property file. It simply outputs a help text and then calls the method *exit()* of the class *System*, which immediately terminates the program. Via parameter 1 the method conveys that it was not a normal program end. A normal program end is signaled with the return value 0.

Listing 28.2 The method »showHelp()« outputs information about the correct program start.

```
26   private static void showHelp() {
27     System.out.println(
28         "Please use two different parameters\n");
29     System.out.println("TransferApp <Source> <Destination\n");
30     System.out.println("or the following properties file:\n");
31     System.out.println("<TransferApp directory>/App.properties\n");
32     System.out.println("with the following entries:\n");
33     System.out.println("Source <Source directory>");
34     System.out.println("Destination <Destination directorys>");
35     System.exit(1);
36   }
```

Methode »main()«

This method implements the process diagram from Figure 28.3. At the beginning, the method creates a String object for the warning that the program issues in case of equal parameters. Then it creates an object of class *Properties* which will read the parameters from a control file in case the user has not entered any parameters from the command line.

Listing 28.3 The method »main()« starts the transfer program

```
38   public static void main(String args[]) {
39     switch (args.length) {
40     case 0: //Case 1: No parameter => try to load the properties file
41       String source = null;
42       String destination = null;
43       try {
44         System.out.println("Installation␣directory:\n\n"
45           + System.getProperty("user.dir")+ "\n");
46         Properties configuration = new Properties();
47         configuration.load(new FileInputStream("App.properties"));
48         source = configuration.getProperty("Source");
49         destination = configuration.getProperty("Destination");
50       } catch (IOException exception) { //Case 1.1: Properties file not
             found
51         System.out.println("Properties␣file␣not␣found:␣" + exception);
52         showHelp(); //Show help
53       };
54
55       if ((source != null) && (destination != null)) {
```

```
56          if (source.equals(destination)) { //Case 1.2: Source =
                Destination
57            System.err.println("Error:␣Source␣=␣Destination\n");
58            showHelp(); //Show help
59          } else { //Case 1.3: Source != Destination
60            new TransferApp(source, destination);
61          }
62        } else {
63          System.err.println("Error:␣Properties␣file␣not␣correct\n");
64          showHelp(); //Show help
65        }
66        break; //Case 1
67
68      case 1: //Case 2: Too few parameters
69        System.err.println("\nError:␣Too␣few␣parameters\n");
70        showHelp();
71        break;
72
73      case 2: //Case 3: 2 Parameters
74        if (args[0].equals(args[1])) { //Case 3.1: Source = Destination
75          System.err.println("Error:␣Source␣=␣Destination\n");
76          showHelp();
77        }
78        new TransferApp(args[0], args[1]); //Case 3.2: Source !=
              Destination
79        break;
80
81      // Exercise on my website:
82      default: //Case 4: Too many parameters
83        System.err.println("\nError:␣Too␣many␣parameters\n");
84        showHelp();
85        break;
86      }
87    }
```

This is followed by the said case decision, which is handled by a case statement. The command line parameters enter the method *main()* through a string array named *args*. Each string array has a method called *length* that can be used to determine its length. If the length is 0, no parameter was passed, if it is 1, a parameter was passed, and so on.

Figure 28.8 shows again the overall flow of the Transfer program in a sequence diagram. After calling the Main function (point 1 of the diagram), the program analyzes the user input (points 1.2 to 1.8) to decide immediately whether it makes sense to start the copy operation. If the user input makes sense, the program creates an object of the class *Transfer*. The constructor of this class creates an object of the class *copy logic* and starts the copy process.

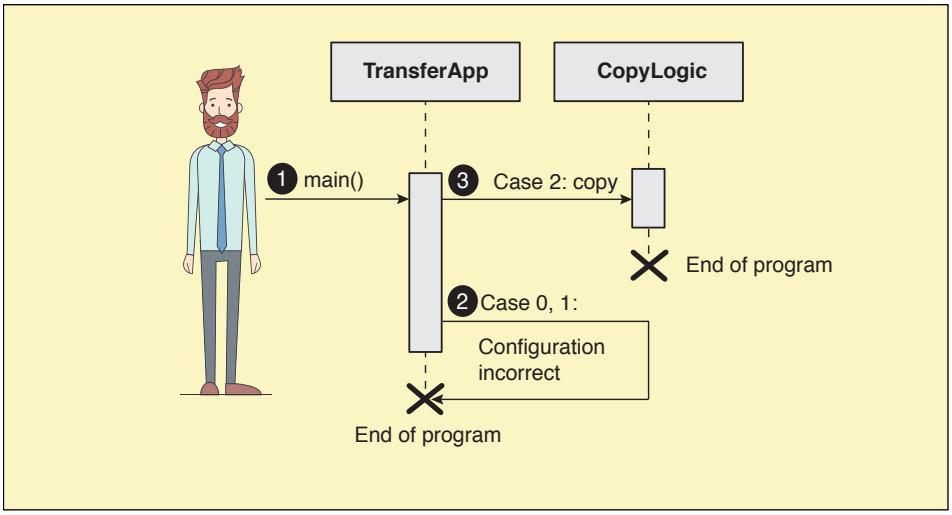

Figure 28.8 The sequence diagram of the overall flow.

28.4.2 Class »CopyLogic«

Package Structure

The class *CopyLogic* is assigned the same package as the main class of the application.

Used Classes

The class *CopyLogic* needs only classes of the Java IO library to copy files and directories. The copy method has only two variables to remember the source and destination directory.

Copy Methods

The copy method *rcopy()* is the core of the program algorithm. It ensures that both files and subdirectories are copied. For this purpose two objects of the type *File* are passed to it. Subsequently, the method determines whether a directory or a file is to be copied. If it is a directory (Case 1), a new directory with the same name is created in the target path, the source directory is read in and converted into an array. The method processes this array recursively, i.e. it calls itself again and again until the entire directory tree has been copied.

Listing 28.4 The method »rcopy« performs the copy operation.

```
38   private static void rcopy(File source, File destination) throws
         IOException {
39     logger.info("Copy from:\n\n" + source + "\n\nto:\n\n" + destination);
40     if (source.isDirectory()) { //Is the source a directory?
41       destination.mkdir(); //Create directory in the destination path
42       String[] directoryEntry = source.list(); // List of directory
             entries
43       for (int i = 0; i < directoryEntry.length; i++) {
```

```
44          String entry = directoryEntry[i];
45          logger.info("\nCopy ".concat(String.valueOf(
46              String.valueOf(entry)))); //Debug info
47          rcopy(new File(source, entry), //Recursively copy all source
                directories
48              new File(destination, entry)); //... to the destination
49        }
50      }
51      else { //No? Then the source is a file!
52        int numberOfBytes;
53        byte[] buffer = new byte[32768];
54        FileInputStream in = new FileInputStream(source);
55        FileOutputStream out = new FileOutputStream(destination);
56        while ((numberOfBytes = in.read(buffer)) > 0) {
57          out.write(buffer, 0, numberOfBytes);
58        }
59        in.close(); //Close file input stream
60        out.close(); //Close file output stream
61      }
62    }
```

By the fact that the method makes a case distinction at the beginning whether a file or a directory is to be copied, the following is achieved with a renewed run: If case 2 occurs and a file is recognized, this is copied. This continues until the method has also copied the last file.

28.4.3 Properties File

To prevent the program from producing errors due to an accidental start, the *source directory* and *destination directory* parameters of the properties file initially have no values after them. If the program is started accidentally, it performs a comparison and issues an error message because both parameters are the same (Case 1.2).

Listing 28.5 The properties provide the parameters for the transfer program.

```
1  #================================================
2  #Program: Transfer
3  #Description: Book "Getting Started With Java Using Eclipse"
4  #Copyright: (c) 2023 by Bernhard Steppan
5  #Autor Bernhard Steppan
6  #Version 1.0
7  #================================================
8  #
9  #========= Correct path for Windows ==============
10 #Source=C:/Source
11 #Destination=U:/Destination
12 #================================================
13 #
14 #===== Correct path for Unix (Linux, MacOS) ======
15 #Source=/source
```

```
#Destination=/destination
#================================================
#
#================= Preferences ==================
#Source
#Destination
#================================================
```

In this way, the user should be forced to consciously edit the file and fill it with meaningful values. Comments are introduced in properties files with a double cross. You can use them, for example, to remember multiple source and destination directories.

28.5 Test

To test the program start the program from the Eclipse development environment. To create a startup configuration, now right-click on the main class *Transfer*. Then select RUN AS → JAVA APPLICATION from the context menu. It makes sense to have several start configurations to check the effect of the case discrimination of the method *main*.

When experimenting with the Transfer program, keep in mind that copy programs can do great damage if they have not been tested sufficiently and carefully. Therefore, work only with directories that do not contain important information until you are sure that the program works without errors.

28.6 Deployment

Deployment is the final step in using a Java program as the user is accustomed to using other programs that come with the operating system, for example. Although the program is small, it can also be delivered in an archive so that the individual components are bundled. To do this, perform archiving with the Eclipse tool (Chapter 20, »Development Processes«).

Listing 28.6 This batch file starts the transfer program without Eclipse.

```
@echo off
REM
REM Programm: Transfer
REM Description: Book "Getting Started With Java Using Eclipse"
REM Copyright: (c) 2023 by Bernhard Steppan
REM Author: Bernhard Steppan
REM Version 1.0
REM
REM Please enter here the correct path to the bin directory of the JDK:
java -jar Transfer.jar
@echo on
```

After that you have to write a batch file (Windows) or a shell script (Unix) for starting the application. How such a batch file looks like can be seen in Listing 28.6).

28.7 Summary

Terminal programs are programs without a graphical user interface. This program has the task to copy a directory tree completely to another place. It proceeds recursively. This means that the copy method calls itself.

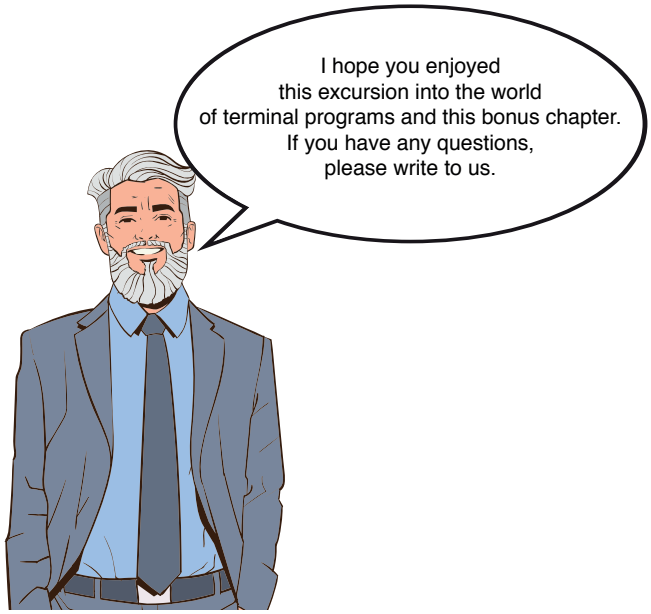

Figure 28.9 If you have any questions, please write to us.

The copy program is a good example of error handling. It is controlled by input parameters. If these are not available, it searches for a batch file. If this cannot be found either, it prints an error message on the terminal.