Poul Klausen

JAVA 1

Basic syntax and semantics

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 1: BASIC SYNTAX AND SEMANTICS

SOFTWARE DEVELOPMENT

Java 1: Basic syntax and semantics: Software Development
1st edition

ISBN 978-87-403-1689-6
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the first in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. The subject of the current book is an introduction to the programming language Java with an emphasis on basic language syntax and semantics, but it is also a book about what programming in general is and how to practically write and test simple programs. The book requires no knowledge about programming or the language Java, and the goal is to show how to get started writing computer programs. After reading the book and worked through the book's exercises and problems, the reader should be able to write simple console applications in the language Java.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

- NetBeans as IDE for application development
- MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
- GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not “how to write” or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

A computer program is a series of commands executed in a certain order, and together they solve a specific task. A program is written as a text document that contains all the necessary commands. This document is called the program code or source code. The individual commands must be written in a very precise way for the computer to understand them, and it is here that programming languages comes into the picture. A programming language lays down precise rules for how the commands should be entered. There are many programming languages, and although they are different, each with their advantages and disadvantages, the similarities outweigh the differences, and once you have learned a language, it is easy to learn the next. The following are used throughout the Java programming language, which is a widely used languages on many platforms. How the individual commands and orders exactly must be written is called the language's syntax. What the individual commands are doing or performing is called the language's semantics.

As mentioned above, a program is written as a text document (in practice several or many), and it is simply a document of commands. Commands are also called statements. These commands or statements being only text, the machine can not immediately perform the comands, but they must be translated into an internal format that the computer understands. This process is called translation or *compilation* and is performed by a program that can convert statements written in a particular programming language to the computer's internal commands. The program is usually called a *compiler*. During the translation the program is checked for errors, and if there are errors, you get an error message, and the errors must then be corrected before the program is translated again. Not all errors are found during translation, but only syntax errors, which covers the issue where a statement is not written in accordance with the programming language's rules. A translated program can easily contain errors, for example a miscalculation.

To write a program you must of course learn the programming language that is selected, but also you must learn how the solution of a task can be formulated by statements in the language. It is the latter that is the most difficult, and there is rarely a clear solution. A solution of a problem by means of a program is also called an *algorithm*. Programming is largely a matter of writing algorithms, something that I will return to several times.

When you have to write software, you need a tool that can be used to enter the program code, and in principle you could use a simple input program (a text editor) and then the compiler, but in practice you will always use a specific development tool, as it makes the job much easier. In the following I will everywhere use *NetBeans*, a development tool for a wide variety of tasks, including writing code in Java. It is an integrated software package, which includes all the tools necessary for the development of a number of different types of programs.

Java is an object oriented programming language. The fundamental architectural element in a program is a *class*, and from the programmer's point of view a Java program consists of a family of classes that collectively define all the application's features and functionality. Writing a program is thus to define – design – and write the code to the program's classes. Nothing in Java exists outside of a class. A program will also always apply other classes that are not written by the programmer, but classes coming from the Java API, and thus is available for the programmer as finished components. One of the program's classes have a special role as the program's "entry point" and the place where the program starts and this class should be written with a special naming, but it is almost the only formal requirements for the architecture of a Java program.

Java is technically both a platform and a programming language. Seen as a programming language, it is a high-level language, which is characterized by

- it is a simple language
- it is an object-oriented language
- the language is architecture neutral
- Java programs are portable
- it supports development of multithreaded applications
- it supports the development of distributed applications
- it supports the development of programs with strong security
- development of effective programs
- development of robust programs
- development of maintenance-friendly and dynamic programs

All Java code are as mentioned written as plain text files – which filename must have the extension *.java* – and then these files are translates to *.class* files. The translation is performed by the Java compiler called *javac*. Java class files do not contain machine code for a particular platform, but rather so-called *bytecode*, which is the machine code for the *Java Virtual Machine*, which is a virtual computer, commonly referred to as *VM* or *JVM*. The program (consisting of a set of class files) can then be carried out by the virtual machine, which is a program that is running on a particular machine. Since Java and thus the virtual machine is available for many different operating systems, the same class files can run on many machines for example Windows, Solaris, Linux, etc.

With a platform wee understand the hardware and software, where a program is running, and in relation to an usual PC you can think of Windows, Linux and Mac machines. Compared to this is a Java platform a software-only solution that runs on a different hardware based platform. The Java Platform consists in principle of two parts:

- JVM, the *Java Virtual Machine*
- an API, the *Java Application Programming Interface*

where the last is a large collection of ready to use software components that a program can use. They are grouped into libraries called *packages*, which consists of classes and interfaces.

The result of the above technology is that Java programs, in principle, is a bit slower than programs translated into an actual physical machine. However, since Java was born there has been an incredible number of improvements and optimizations of both the compiler and the virtual machine, so the difference in performance is negligible if at all measurable.

2 HELLO WORLD

The subject of this chapter is to show how to write and run a Java program using NetBeans and the aim is solely to get started. There are several kinds of programs, or you can say that the programs can be categorized in several ways, but in the first books I will look at three types of programs:

- *commands*, which is a program that is typically performed from a command window (a *Terminal*) where you enters the program's name that may be followed by one or more arguments
- *consol applications*, which also is performed at a command prompt, but here the program runs in a dialogue with the user, where the user must enter values when the program is executed
- *GUI-programs*, where the program opens one or more windows with components as buttons and input fields used by the user to interact with the program

This division is only for practical reasons, as I will sometimes characterize the program examples in relation to this, but common to the three types of programs is that they are standalone applications that run on a single machine and without using resources on other machines.

I'll start with the classic *Hello World* program, a program that prints a text on the screen. It is an example of a command, but it is also an example of a program that has absolutely no practical interest. Although it is a simple program, it will nevertheless treat a number of basic principles that apply to all Java programs.

2.1 NETBEANS

As mentioned a Java program is written as text files, which will then be translated. When the files are translated without errors the program can be executed by the virtual machine. In practice is always used a development tool, which is a program or software package that integrates all the functions that a developer needs. Such a tool is usually called an IDE (*Integrated Development Environment*), and there are several, but I will everywhere use *NetBeans*, which contains everything that is needed, and the following requires that both Java and NetBeans are installed on the machine. Do not have it, you can start by reading Appendix A which explains how to download and install both Java and NetBeans.

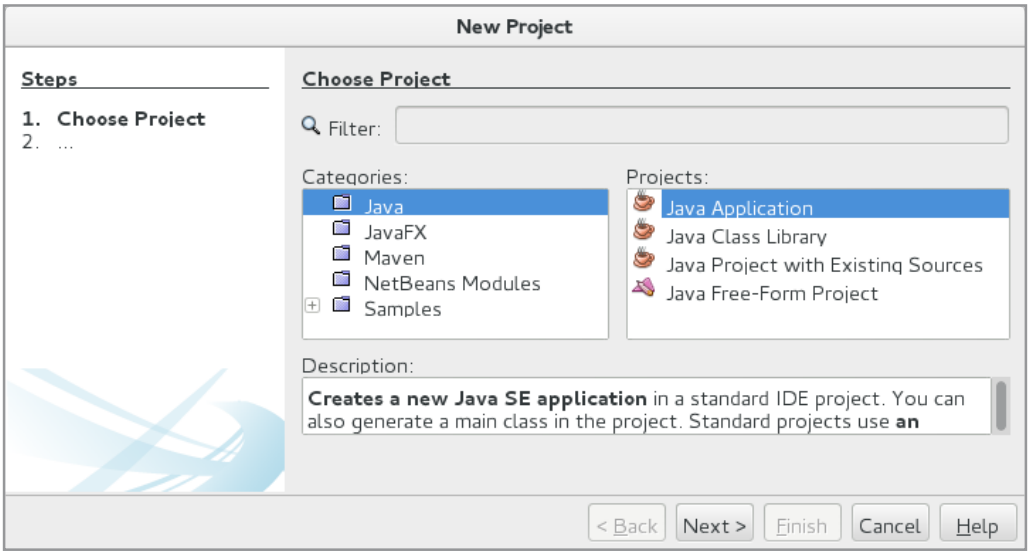
To write the first Java program, open NetBeans and create a new project. In the menu, choose

File | *New Project* (see below)

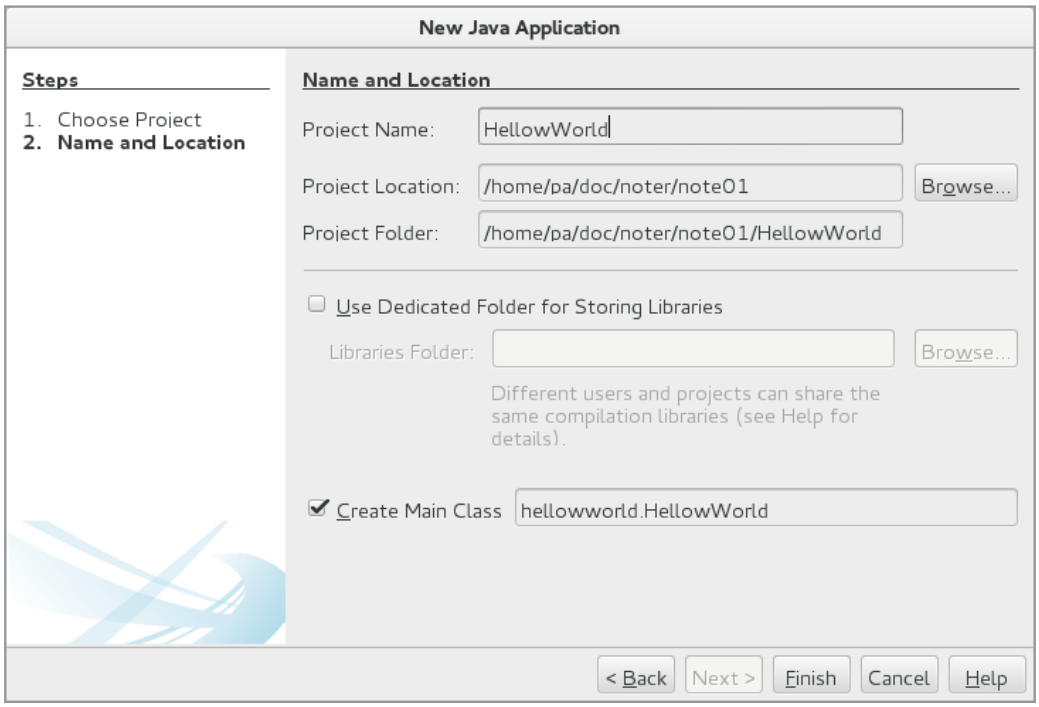
A NetBeans project creates all the necessary files required to develop and test the program and eliminates a variety of configurations that are otherwise necessary. Using NetBeans you can build and run the program just by clicking the menu. To create the project, you must note that in this case,

- *Java* is marked in *Categories*
- *Java Application* is selected in *Projects*

but otherwise I have not done anything in the first window.



When you then click Next you get the following window:



Here I:

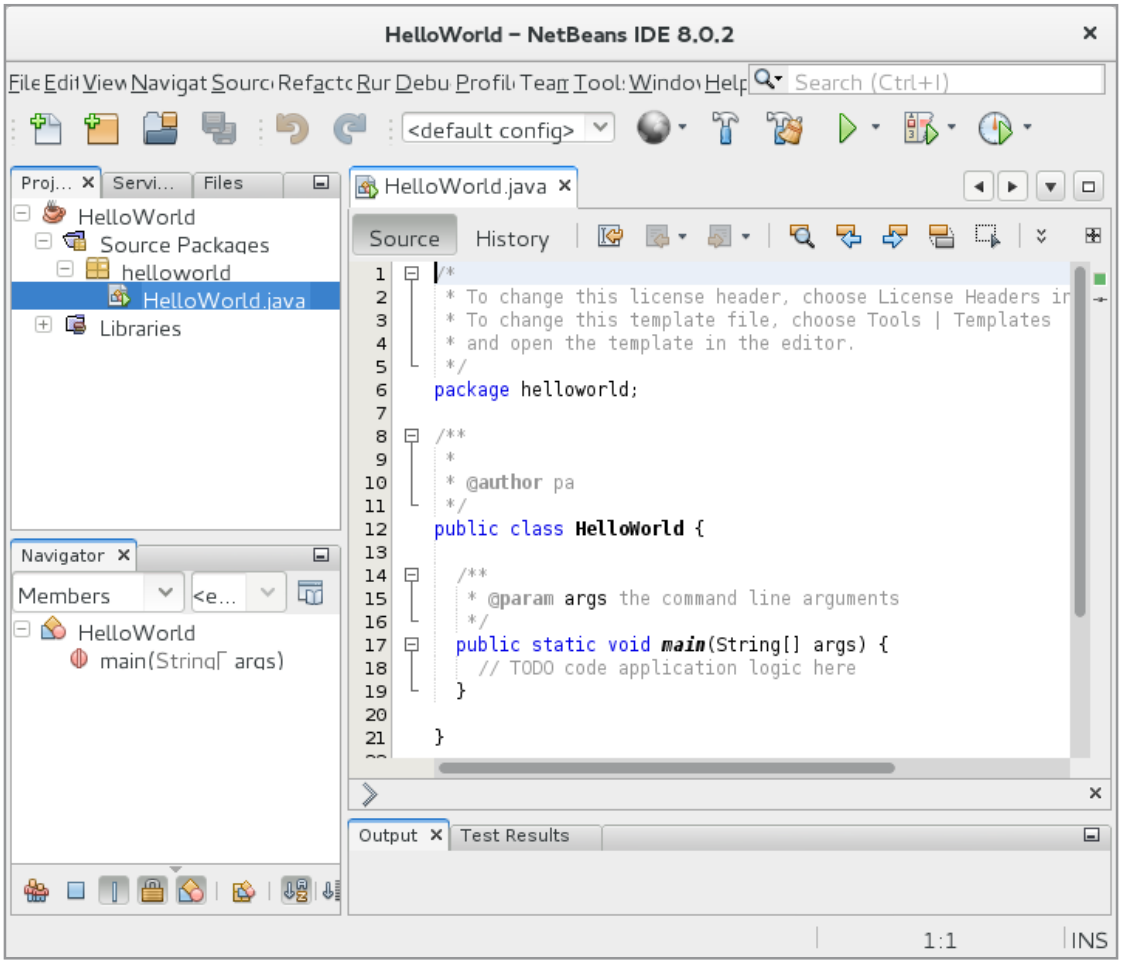
- Entered *HelloWorld* as *Protect Name*
- Selected */home/pa/doc/noter/note01* for *Project Location*

Regarding the latter, it is just a question that I have decided that the project should be created in the */home/pa/doc/notes/note01* direcory. I have also decided that the project should be called *HelloWorld*. NetBeans will then create a folder

/home/pa/doc/noter/note01/HelloWorld

and all project files are placed by NetBeans in this folder. You should note that there is a checkmark in the *Create Main Class*, which is important.

When you then click Finish, the project is created and NetBeans displays multiple windows:



- *Projects* window, where all the project's components are organized in a hierarchy
- *Source Editor*, where there is an open file called *HelloWorld.java*
- *Navigater*, which can be used to quickly find a specific item

Source Editor contains the program's code, and as you can see, NetBeans automatically creates a skeleton for a program. Actually, it's a full-fledged program – it performs nothing not yet. The program code consists of Java statements and comments. Comments are removed by the compiler and does not affect the finished program. They are inserted solely for the sake of us people who should read and understand the program code.

2.2 THE SOURCE CODE

Below I've shown the finished program after I have changed or removed the comments that NetBeans has generated and written a single statement:

```
/*
 * A simple Java program that prints a text in a console window.
 */
package helloworld;

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

There is only one comment back, which is at the top and says something about what the program does. The rest of the code is Java statements. A Java program consists of classes, and in this case there is one class called *HelloWorld* and thus has the name that I chose as project name. The class consists of a *method* called *main()*, which has a single statement – not created by NetBeans, but as I entered. It is a statement that prints a text on the screen.

The code is simple, and so far you just accept that it should be written, as shown above, but there is however a few things that you should note.

Java is case-sensitive, so everywhere you must distinguish between uppercase and lowercase letters.

Every Java program consists as already mentioned by at least one class here called *HelloWorld*. A class consists of *variables* and *methods*. In this case, the class has only one method called *main()*, which is the method that is called when the program starts. A method consists of statements that can be perceived as commands that perform one or other on the machine. That a method is called means that the methods statements is performed. Note that the method *main()* must be prefixed by the words *public static void*. The explanation will follow. In this case, *main()* has only a single statement, that write a text on the screen. *System.out.println()* is actually a method in a class *PrintStream*, that among other things, represents the screen. When the program is running, nothing happens than the *println()* statement in *main()* is performed that prints a text on the screen.

Note that in Java, each statement ends with a semicolon – above there is a semicolon after *System.out.println()*. It tells the compiler where a statement ends.

In Java classes are grouped in so-called packages. A class's full name consists of the package that the class it is grouped under, as well as the class name. NetBeans automatically define a package for a program that is the application name written in lowercase, and the first statement is a package statement indicating the class's package. A package statement must be the first statement in the file that contains a class, but can be prefixed by a comment. Note that in Java you generally has to place each class in its own file, but more on that later.

2.3 RUN THE PROGRAM

When the program is written as above (without errors), you can from the menu in NetBeans choose

```
Run | Run Project (HelloWorld)
```

NetBeans will automatically translate the program, and if it not contains errors, the program will be performed:

```
run:
Hello World
BUILD SUCCESSFUL (total time: 0 seconds)
```

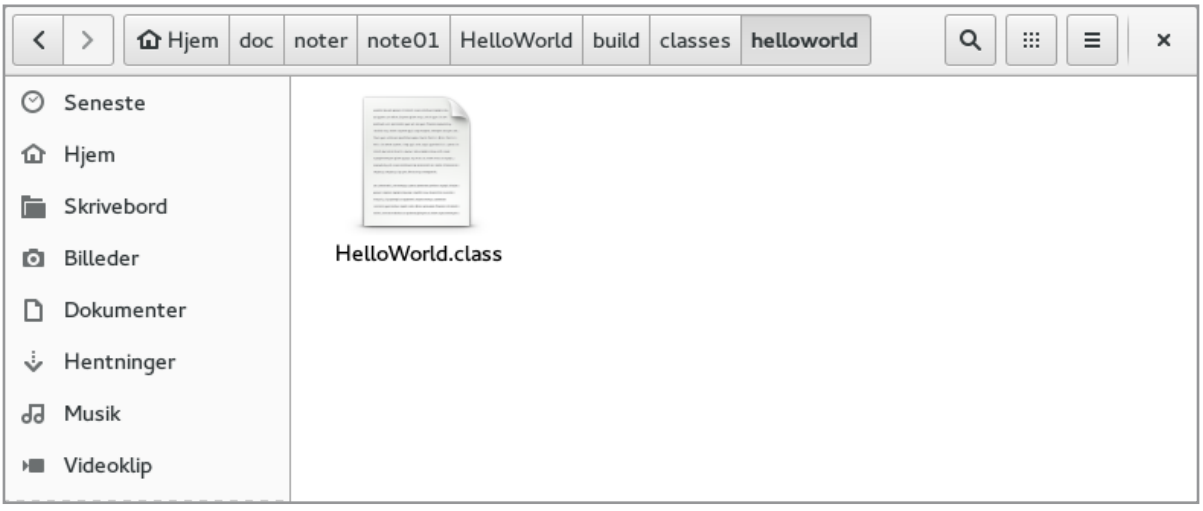
If the program contains errors, the result could be the following:

```
run:
Exception in thread "main" java.lang.RuntimeException:
Uncompilable source code - Erroneous tree type:
   <any> at helloworld.HelloWorld.main(HelloWorld.java:10)
Java Result: 1
BUILD SUCCESSFUL (total time: 3 seconds)
```

The program then is not performed, and the translator instead offers an error message, and the error must then be corrected before trying to run the program again.

2.4 THE NETBEANS PROJECT

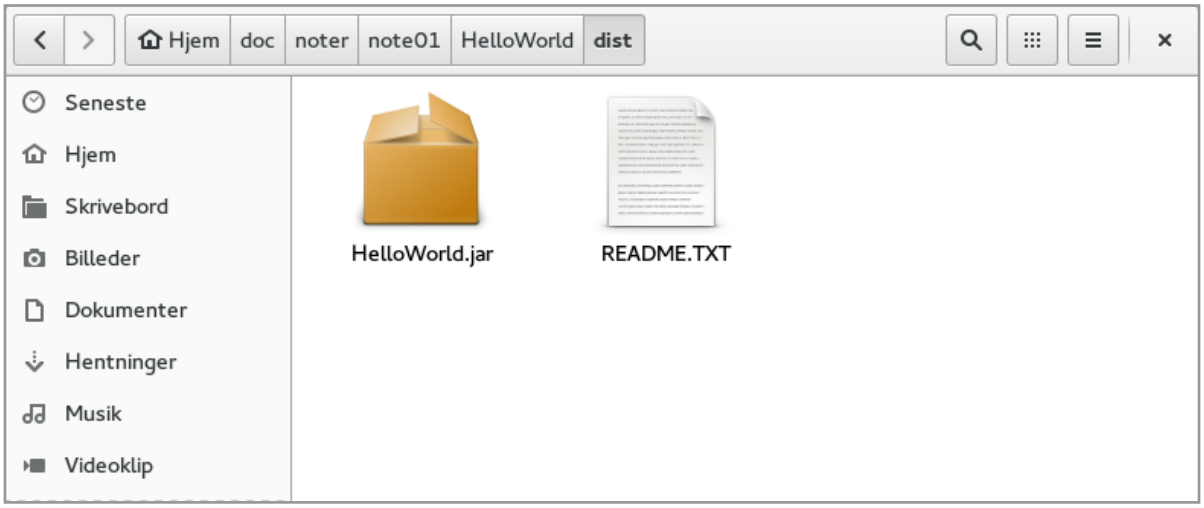
If you open *Files* you can as shown below find the *class* file, which is the translated program:



HelloWorld.class is an example of a complete Java program that can run on a specific machine. Apparently, the program is tied closely to NetBeans and performed using NetBeans, but it is not the case. In the menu in NetBeans choose

Run | Build Project (HelloWorld)

and the program is translated again, and if you opens *Files* you will discover that that there is created another folder called *dist*, which contains two files (see below). Here is *HelloWorld.jar* a package (actually a compressed zip file) containing the program files – only the translated class files and other ancillary files that are necessary for the program to run. In this case, there are actually only two.



If you copy the file *HelloWorld.jar* to a folder – for example *temp* – and if you open a *Terminal*, stand in this folder and perform the following command:

```
java -jar HelloWorld.jar
```

the program is performed:

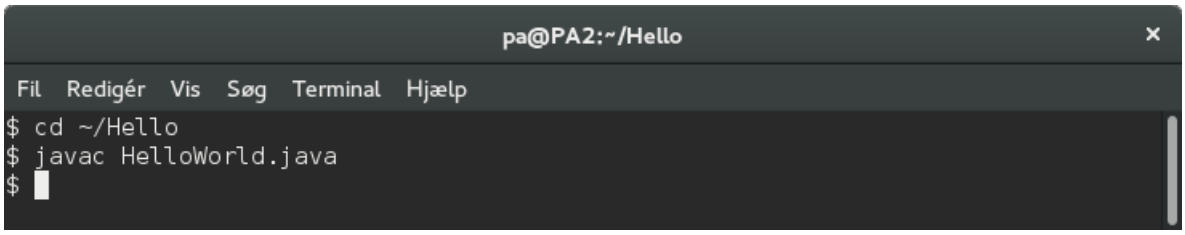
Here the command *java* is a message to Linux to start the *Java runtime system* and execute the program *HelloWorld*.

2.5 GEDIT

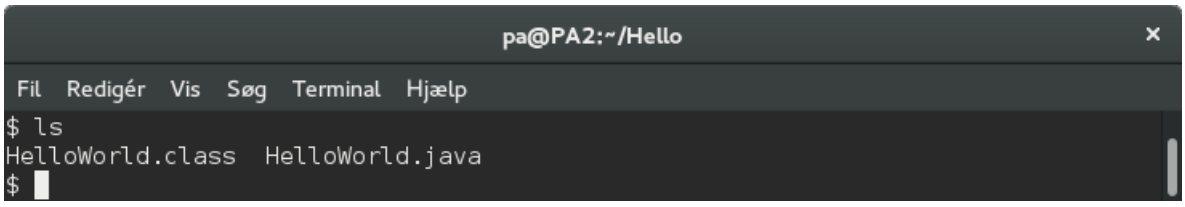
Above I have written and performed a Java program using NetBeans, and in the following, all programs will be developed in this way. NetBeans is a large and complex program, and until this place, you have seen only a very small fraction of what the program can. The program is relatively user-friendly, and I will not give any general description of the program and its possibilities, but I will mention important features as I need them in the individual examples. NetBeans is similar to all other IDEs for software development, so the forces you use to learn more about the program, is definitely not wasted. Using an IDE for developing applications provides in practice such large benefits that it makes no sense to develop programs in other ways, but in principle you can, and I will in this section show how to write the program *Hello World* without NetBeans.

I start by creating a directory named *Hello* in my home directory. Then I open *gedit* and enter the code below. It is important to enter the code exactly as shown below, and especially you should be aware that it is case-sensitive. You should also be aware that you do not insert extra spaces. Note also that the text is displayed in several colors. This is because *gedit* know Java, and thus highlights reserved words.

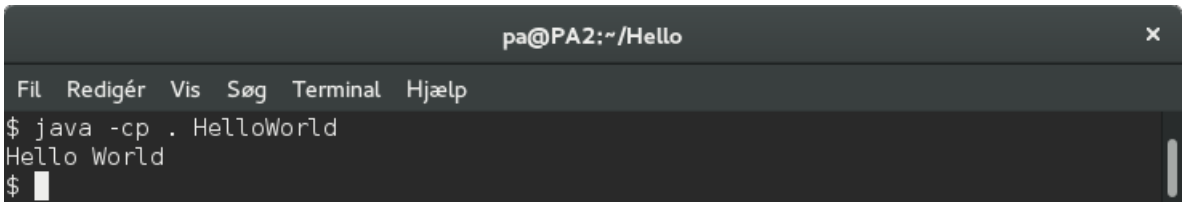
When the program is written, I save it in the folder *Hello* and call it *HelloWorld.java*. Again, please note upper and lower case and the filename must have the extension *java* written in lower case. Then I open a *Terminal* and set the current directory to the folder *Hello*. Here I performs the following command:



javac is the name of the compiler, and the result is that the program is being translated, and creates a file with the translated program called *HelloWorld.class*:



Then you can execute the program as follows:



As mentioned, I practically always use NetBeans, and even if the above is not of great practical interest, it may nevertheless serve to illustrate what NetBeans is doing. It is basically an advanced editor program that help you when you enter code and point out incorrect entries. Moreover NetBeans calls both the *javac* compiler and the runtime system *java*. NetBeans will provide support to the programmer and support the development of large and complex applications. Therefore even small NetBeans projects contains many directories and files, and it can hide what really is the program itself. This example can show how little it really is.

2.6 SOMETHING ABOUT COMMENTS

As you have seen, NetBeans inserts comments in the program's code. They will as mentioned be ignored by the compiler and are inserted for the sake of the people who must read and understand the code.

There are three types of comments. The first starts with /* and ends with */ and everything in between these character combinations is considered as a comment, for example

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
```

which is the comment that NetBeans inserts at the beginning of a new source file. Another kind of comments start with the characters /** and ends with */, for example

```
/**
 * @param args the command line arguments
 */
```

Again, everything between the start and end character combinations is comments, but there can be inserted special symbols that are interpreted by a tool to generate html documentation of the program code, but about that later. The last type of comment has the form

```
// TODO code application logic here
```

That comment can be inserted anywhere, and all after the characters // to the end of the line are considered as a comment.

This is the syntax for inserting comments in the code, but a whole other thing is what you should write. Also I come back to that later, but generally you have to write what you think might be valuable at a later reading of the code. What it is, is certainly not unique, and it can be good inspiration to examine what comments others have inserted into programs.

2.7 EXAMPLE: KINGS

The example is a program called *Kings*, in principle it is written in the same way as *HelloWorld*, but the program prints the following text:

```
#################################
# Gorm den gamle  #       #  958   #
# Harald Blåtand  #       #  987   #
# Svend Tveskæg   #       #  1014  #
# Harald d. 2.    # 1014  #  1018  #
# Knud den Store  # 1018  #  1035  #
#################################
```

The final code is shown below:

```
package kings;
public class Kings
{
 public static void main(String[] args)
 {
```

```
  System.out.println("###################################");
  System.out.println("# Gorm den gamle #       #  958   #");
  System.out.println("# Harald Blåtand #       #  987   #");
  System.out.println("# Svend Tveskæg  #       #  1014  #");
  System.out.println("# Harald d. 2.   # 1014  #  1018  #");
  System.out.println("# Knud den Store # 1018  #  1035  #");
  System.out.println("###################################");
 }
}
```

The only difference compared to *HelloWorld* is that this time there are several *System.out.println()* statements.

EXERCISE 1

Write a program, as you can call *Digits*, that on the screen prints the following table:

```
*************
* 1 * One   *
* 2 * Two   *
* 3 * Three *
* 4 * Four  *
* 5 * Five  *
* 6 * Six   *
* 7 * Seven *
* 8 * Eight *
* 9 * Nine  *
*************
```

EXERCISE 2

Write a program that you can call *Label*, that prints your name, your address and your email address, for example

```
Poul Klausen
Tjørnevænget 56
7800 Skive
poul.klausen@mail.dk
```

3 COMMANDS AND CONSOLE PROGRAMS

In the previous section I divided the programs into three categories, and in this chapter I will look at commands and console programs. In principle there is no big difference, and the division alone has to do with how the user is transferring data to the program. Both *HelloWorld* and the example *Kings* from the previous chapter are examples of commands.

3.1 COMMANDS

Every Java program must have a *main()* method that has the following signature:

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

For the moment you should ignore the meaning of the words *public*, *static* and *void* and just accept that they should be there, but the *main()* method is the place where the program starts. After the method name is a parameter in parentheses, indicating arguments from the command line that can be transferred to the program. Consider the following program:

```
package command;

public class Command
{
 public static void main(String[] args)
 {
  System.out.println(args[0]);
  System.out.println(args[1]);
 }
}
```

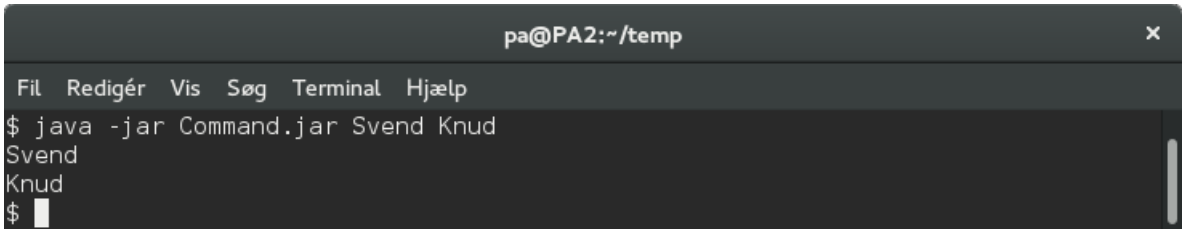
args is an array, as I explains later, but the arguments that are transferred on the command line, are in the program referred to as

args[0]

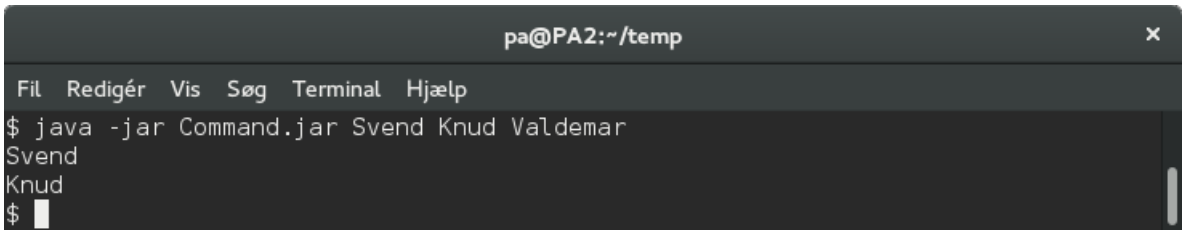
args[1]

args[2]

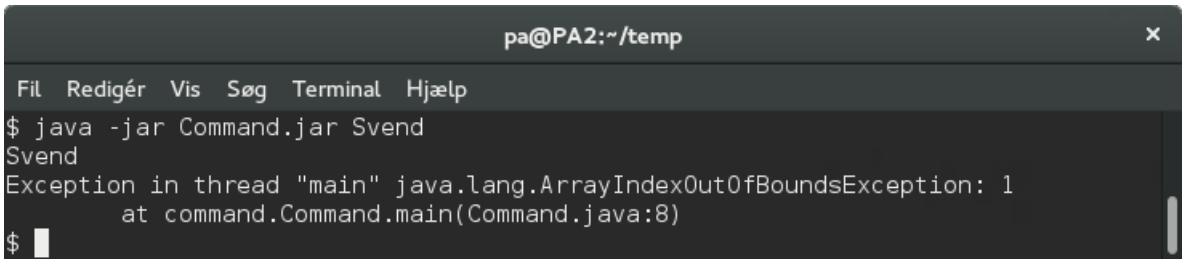
and so on. An argument is a text string, and arguments are separated by spaces. If, for example you copy the file *Command.jar* to the *temp* folder, the program can be perform as follows:



There are two arguments, called respectively *Svend* and *Knud*, and the program prints the two arguments of the screen. If you execute the program in the following way:



the result is the same. This time there is three arguments, but only the first two are used in the program. If, however, the application performs without having two arguments, you get an error:



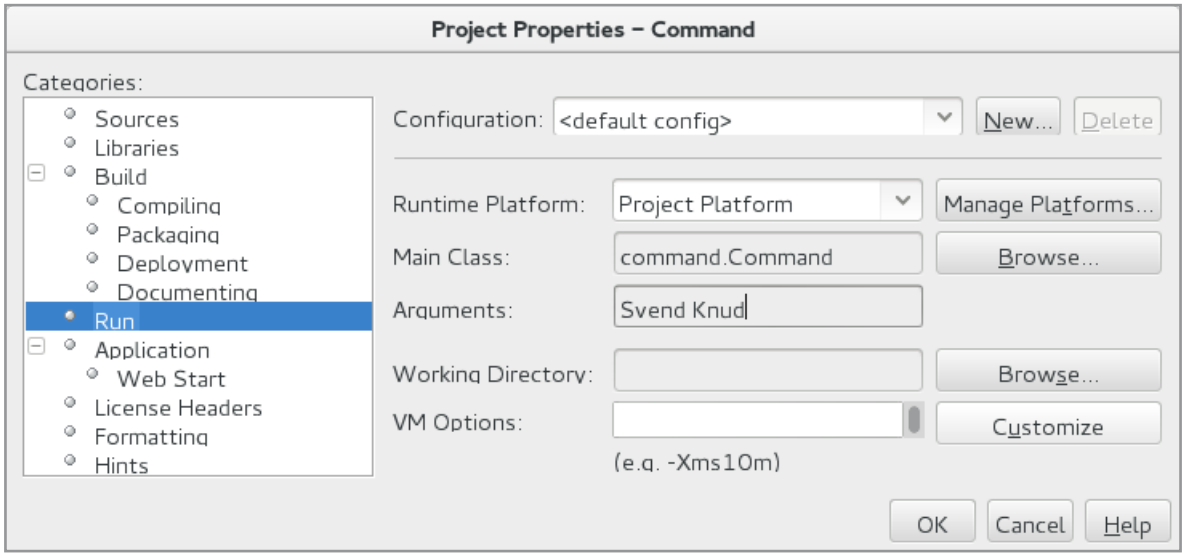
The reason is that the second argument does not exist, and therefore fails the statement:

```
System.out.println(args[1]);
```

The example shows, what I will understand by a command. Another question is how to test the application from NetBeans when you has to transferre arguments. This can be done through the menu to choose

```
File | Project Properties (Command)
```

Here you must select the category *Run*, where you will be able to enter the arguments:



Note that contains the argument spaces, it is necessary to specify them in quotes.

3.2 EXAMPLE: PRINTADDRESS

The following program will print the name and address of a person, but so that the values to be printed are transferred on the command line. The program is therefore a command. The program is called *PrintAddress*, and threre must be transfered four arguments on the command line. Arguments on the command line are separated by spaces, and if an argument contains spaces, it is necessary to put the argument in quotes. Then the runtime system will perceive it as one argument. An example of running the program could, for example be as shown below:

```
pa@pa3:~/temp
Fil  Redigér  Vis  Søg  Terminal  Hjælp
$ java -jar PrintAddress.jar "Poul Klausen" "Tjørnevænget 56" 7800 Skive
Address:

   Name:    Poul Klausen
   Address: Tjørnevænget 56
            7800 Skive
$
```

For writing the program I have in NetBeans created a project called *PrintAddress*. The program code can then be written as follows:

```
/*
 * Program that on the screen printer a person's name and address.
 * The values are passed as arguments on the command line.
 */
package printaddress;

public class PrintAddress
{
 /*
  * Enter a person's name and address on the form
  *  name address zipcode town
  * that is four fields separated by spaces.
  */
 public static void main(String[] args)
 {
  System.out.println("Address:");
  System.out.println();
  System.out.println("  Name:   " + args[0]);
  System.out.println("  Address: " + args[1]);
  System.out.println("       "+ args[2] + " " + args[3]);
 }
}
```

The statements in the *main()* method is all *System.out.println()* statements that prints a line on the screen. The first only prints a text, while the second prints a blank line. The third prints a text followed by the value of *args[0]*, which is the person's name. Note particularly the plus operator, which means *string concatenation*, wherein the text is added after the other. The last two *System.out.println()* statements works in principle in the same way.

A program like the above are not robust, as it will fail if not transferred the right number of arguments. You should also note that the program does not test the arguments (which incidentally is also not so easy), but simply prints the arguments as they are.

PROBLEM 1

You should write a program for a library that can print a recall of a book. When the program is executed, you must on the command line transfer five arguments:

- borrower's name
- borrower's address
- borrower's zipcode and town
- ISBN of the book
- the books title

An example of an execution of the program might be:

```
                ############
                #  Recall  #
                ############

To:

  Poul Klausen
  Tjørnevænget 56
  7800 Skive

When we can see that the loan period for the following book is the end, we
must ask you as quick as possible return the book to the library.

Title:

  978-0132126953
  Computer Networks

Yours sincerely
The Research Library
```

3.3 CONSOLE PROGRAMS

Compared to a command a console program (in this books) is a program that performs a dialogue with the user, where the user must enter data.

Above I have shown a program that prints the name and address of a person when the values are passed as arguments on the command line. Below is the same program, but this time the user must enter the values during running the program. There is thus a dialogue with the user.

```
package inputaddress;

import java.util.*;

public class InputAddress
{
 public static void main(String[] args)
 {
  Scanner in = new Scanner(System.in);
  System.out.print("Enter name:  ");
  String navn = in.nextLine();
```

```
    System.out.print("Enter address: ");
    String adres = in.nextLine();
    System.out.print("Enter zipcode: ");
    String postnr = in.nextLine();
    System.out.print("Enter town:   ");
    String bynavn = in.nextLine();
    System.out.println();
    System.out.println("Address:");
    System.out.println();
    System.out.println("  Name:     " + navn);
    System.out.println("  Address:  " + adres);
    System.out.println("            "+ postnr + " " + bynavn);
  }
}
```

As you can see, the code has been substantially larger, and there is also new things that has to be explained.

To enter text, you must have an object that represents the keyboard and provides a service available for entering text. In addition I apply a *Scanner* that use the objekt *System.in* that just represents the keyboard. *Scanner* is a class that is not readily available, and therefore there is added an *import* statement that refers to the package containing the class *Scanner*. *in* is then an object that can be used to enter a text. The next line prints a text on the screen, telling the user to enter the name. It happens with

```
in.nextLine()
```

The text that the user enters, must be stored somewhere, and for that purpose a *variable* is used. Variables are considered in the next chapter, but a variable is a place where you can store a value. An example could be the statement

```
String name = in.nextLine();
```

Here, the text that the user enters is stored in the variable *name*. The next statements are identical in principle and are used for entering other values. The last statements are used to print the result and is similar in principle to the previous version of the program, but the arguments *args[0]*, *args[1]*, … are replaced with variables. If you run the program (from NetBeans), the result could be as shown below:

```
Enter name:     Poul Klausen
Enter address:  Tjørnevænget 56
Enter zipcode:  7800
Enter town:     Skive

Address:

  Name:      Poul Klausen
  Address:   Tjørnevænget 56
             7800 Skive
```

Today it is rarely – if ever – developing console applications, and when is a need for a program with a user dialogue (and it is of course often), you write a Windows or GUI program. Console programs may, however, for testing and learning be useful, and therefore it is excellent to know how to write a simple console program.

PROBLEM 2

You must solve the same task as in problem 1, but instead of transferring values as arguments on the command line, you must enter information about

borrower's name

- borrower's address
- borrower's zipcode and town
- ISBN of the book
- the books title

when then the program is running in a dialogue with the user. The program should print the same recall as in problem 1.

4 VARIABLES AND DATA TYPES

Programs has to deal with data, and for that they need a way where to save or store data. To that purpose programs use variables that are items, where the program may store a value. You must note that I have already used variables associated with data entry. A variable is characterized by

- a name
- a type
- operators

Variables must have a name, so you can refer to them in the program. Java is similar to other modern programming languages relatively flexible in terms of naming variables, but the following shall (should) be met:

- a variable name must start with a letter, a dollar sign '$' or an underscore '_'
- a variable name should always start with a lowercase letter, and one should avoid '$' and only occasionally use '_'
- then have to follow any number of characters consisting of letters, digits, $ and _
- a name must not contain spaces
- the name of a variable must not be a reserved word, that is a word that has specific meaning in Java

In addition a variable name should tell something about what it is used for. Use whole words instead of cryptic abbreviations. It will make the code easier to read and understand. If you have long names consisting of several words, you can increase readability by letting a word (except the first) start with a capital letter such as *customerAddress*. Alternatively, is there anyone who writes *customer_address*, but generally avoid very long names.

If you do not break these rules, you have never problems with names of variables, but some other characters are actually allowed.

Variables has a type that indicates which values can be stored in them, and how much a variable is filling in the machine's memory. The type determines simultaneously the operations that can be performed on a variable, that is what to do with it.

Variables must be created or declared before they can be used. This is done by a statement of the form:

type name = value;

That is, to write the type first, then the variable name and finally assign it a value, for example

```
int number = 23;
```

Here is declared a variable called *number*, that has type *int* and the value 23. Variables should always be initialized, else you can get a translation error.

When variables must be declared, it is because the compiler must allocate space in the machine's memory, and that when the name appears somewhere in the code, the compiler must know what the name mean to check if the variable is used in the proper context. If not, the compiler will come up with an error message. The program can only be used when it is compiled without errors.

Java has the following built-in primitive or simple data types:

- *byte*, which is a data type for an integer. A variable occupies 8 bits, and may contain values from -128 to 127 (both inclusive).
- *short*, which is a data type for integers. A variable takes up 16 bits and may contain values from -32,768 to 32,767 (both inclusive).
- *int*, which is a data type for an integer. A variable occupies 32 bits and can contain values from -2147483648 to 2147483647 (both inclusive). It is the default type for an integer.
- *long*, which is a data type for an integer. A variable occupies 64 bits and can contain values from -9,223,372,036,854,775,808 to 9223372036854775807 (both inclusive).
- *float*, which is a data type for floating point numbers and can thus be used for decimal numbers. A variable occupies 32 bits and can represent decimal numbers with 7–8 significant digits. It is important to note that the value is always a rounded result.
- *double*, which is a data type for floating point numbers and can thus be used for decimal numbers. A variable occupies 64 bits and can represent decimal numbers with 14 significant digits. It is important to note that the value is always a rounded result. This type is the default type for a floating point.
- *boolean*, which is a data type with only two values: *false* or *true*. The type is important to be able to write conditions and is used specially for program control.
- *char*, used for characters, and a variable of the type *char* takes up 16 bits. Values are numeric codes with values from 0 to 65535 (both inclusive), and each character is represented by a numeric codes. For example has a large A the code 65.

The smallest unit you can use on a digital computer is a *bit*, and the whole computer's memory consists of a number of devices that can store one bit. One bit can represent one of two values, commonly referred to as 0 and 1, and the contents of the computer's memory is always a large number of 0s and 1s. In practice, you cat not directly refer to the individual bits, but they are organized in groups of 8 bits, and such a group is called a *byte*. A byte is a pattern consisting of 8 bits, for example

```
00111001
```

What that means depends on how a program uses the pattern. Since each of the 8 places in a byte, may have two values, a byte therefore can represent different values. As is clear from the above the simple data types are different in how many bits they use to a variable of that type. For example uses an *int* 32 bits (or 4 bytes), and this means that one can represent different integers that is interpreted as the numbers starting from -2147483648 to and including 2147483647.

In addition to the eight primitive data types mentioned above, there is, as already mentioned a type *String* which can contain any text string. For example you can write

```
private String name = "Knud den Hellige";
```

The statement creates a variable of the type String. You should note that you specify the value in quotes. The type *String* is not a primitive type, but it is instead a *class*, but for now you can ignore it, and variables of the type *String* is used in principle in the same way as other variables.

Primitive variables are assigned a default value of the compiler. The type *boolean* have the default value *false*, a *char* has a space as the default value, while the other primitive types have the default value 0. A *String* (which is not a primitive type) has no value, which is defined as *null*.

As an example, the following statements creates three variables, all of the type *int*:

```
int number1 = 17;
int number2 = 23;
int sum = number1 + number2;
System.out.println("The sum of " + number1 + " and " + number2 + " is " + sum);
```

The first two variables are initialized with numbers, while the last is initialized to the sum of the first two. The last statement prints the values of the three variables.

Consider as an example the following program, which is a command to be performed on the command line with three arguments which must be integers, and the program prints the sum of the three numbers:

```
package sum3;

public class Sum3
{
 public static void main(String[] args)
 {
  try
  {
   int a = Integer.parseInt(args[0]);
   int b = Integer.parseInt(args[1]);
   int c = Integer.parseInt(args[2]);
   System.out.println(a + " + " + b + " + " + c + " = " + (a + b + c));
  }
  catch (Exception ex)
  {
   System.out.println("Ulovlige argumenter");
  }
 }
}
```

If you execute the program with arguments 13, 17 and 19, the result will be:

```
13 + 17 + 19 = 49
```

Arguments for a command or data that is entered into a console application, has always the type *String*, and they can not be directly used in the calculations. It is first necessary to convert the values (which is text) to a numeric value. It can, for example be done in the following way:

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

Here, the value of *args[0]* will be converted to an integer – an *int* – and is stored in the variable *a*. In order that it is possible, *args[0]* must contain a legal integer. Is it not the case (because the user has specified an illegal value), then the conversion will raise an exception, which means that the conversion is interrupted with an error. When you have code that can raise an exception, the code must be placed in a *try* block. If so and there is an exception the runtime system interrupts the *try* block, and the control is transferred to the subsequent *catch* block. In this case, it means that if one of the three arguments can not be converted to an *int*, then the program jumps to the *catch* part and prints an error message.

If the three arguments can be converted correctly the result is printed as follows:

```
a + " + " + b + " + " + c + " = " + (a + b + c)
```

This is a relatively complex expression, and may not be easy to read, but for the understanding of the expression, it is easiest to compare with the result:

```
13 + 17 + 19 = 49
```

The numbers 13, 17 and 19 are the values of the three variables *a*, *b* and *c*. The + signs in quotations is just text and included in the results as strings. If you for example consider

```
a + " + "
```

the + sign means string concatenation of the value of the variable *a* and the value of the text " + ", which is a + sign with a gap on both sides. The last parentheses in the result expression

```
(a + b + c)
```

calculates the sum of the three variables, and from that the value 49.

EXERCISE 3

You should write a program similar to the above, but there must be the following differences:

1. The program should calculate the sum of four numbers instead of three numbers
2. The numbers should this time be entered (in a dialogue) instead of passed as arguments on the command line – it must therefore be a console application
3. The numbers should this time be decimal numbers instead of integers – i.e. the type must be *double*

Above is shown how the method *Integer.parseInt()* can convert a text to an *int*. To convert a text into a *double*, you can correspondingly apply the method *Double.parseDouble()*.

An example of running the program could be:

```
Enter the 1. number: 3.25
Enter the 2. number: 1.75
Enter the 3. number: 9.85
Enter the 4. number: 2.55
3.25 + 1.75 + 9.85 + 2.55 = 17.4
```

Note especially that the decimal point is dot.

4.1 OPERATORS

The operators are special symbols known by the compiler, that perform specific operations on one, two or three operands, which are typically values, or variables. Above I have already used the + operator, as an operator between strings that act as string concatenation, while the operator between numbers is interpreted like addition. Also a declaretion in which a variable is assigned a value, the equating is an operator.

The operators have different priorities, which affect the order in which they are evaluated if an expression contains multiple operators, and the following table shows Java's operators in order of priority with falling priority down. If an expression contains multiple operators, the operators with the highest priority are evaluated first. Operators with the same priority are evaluated from left to right. It does not apply to the assignment operators, which is evaluated from right to left.

Priority	Operators
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multplicative	* / %
additive	+ -
shift	<< >> >>>
realtional	< > <= >= instanceof
comparing	== !=
bitvis AND	&
bitvis XOR	^
bitvis OR	|
logisk AND	&&
logisk OR	||

Priority	Operators
question operator	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Many of these operators require no special explanation, and others, I first will deal with later, but below are a few comments on some individual operators.

THE ASSIGNMENT OPERATOR

Note first that assignment is an operator, for example

```
int a = 2;
a = 3;
```

It is undisputed the most commonly used operator.

THE ARITHMETIC OPERATORS

Multiplicative and additive operators generally referred to as arithmetic operators and covers operators for the four arithmetical operations. Furthermore, there is the modulus operator, which is closely linked to division. All five operators have two arguments. If the arguments are integers, the result is again an integer. This means the division is integer division. For example is

```
23 / 7 = 3
```

as the division goes a 3 time and the rest is thrown away. Modulus is the rest with division, and such is

```
23 % 7 = 2
```

when 23 divided by 7 gives the rest 2. Taking the following statements

```
System.out.println(23 / 7);
System.out.println(23 % 7);
System.out.println(-23 / 7);
System.out.println(-23 % 7);
System.out.println(23 / -7);
System.out.println(23 % -7);
System.out.println(-23 / -7);
System.out.println(-23 % -7);
```

they will print

```
3
2
-3
-2
-3
2
3
-2
```

Here, the sign is not quite obvious, but for two integers *a* and *b* are

```
a % b = a - round(a / b) * b
```

where *round(a / b)* means *a / b* with any decimal places thrown away. From this formula, it is easy to figure out that the above results are correct. The modulus operator can also be used if the arguments are floating-point numbers, and the formula is the same, but the result has rare interest in this case. The arithmetic operators can be combined with the assignment operator. As an example means

```
a += 2;
```

the same as

```
a = a + 2;
```

Although it is not an arithmetic operator, one should note that the + is also used for strings, and for example means

```
String s = "Hello" + "World";
```

string concatenation and the value of the variable *s* is *HelloWorld*. This means that if the type of at least one of the operands are *String* the operator + means string concatenering and not addition.

UNARY OPERATORS

There is a family of operators which takes only one argument, and that includes the operators to sign, negation and the two special oprators ++ and --. Finally there is also the operator for binary complement, which I will not mention in this place. A common feature of these operators is that it is the operators with the highest priority. Consider as an example the following statements:

```
int a = 2;
System.out.println(++a);
System.out.println(a++);
System.out.println(a);
```

If done, you get the result:

```
3
3
4
```

The variable *a* has the value 2. Executes ++*a*, it means that the value of the variable is incremented by 1, after which the value is printed. The result is therefore 3 (the variable *a* has the value 3). Executes then *a*++, this means that the value of *a* is printed, after which the variable *a* is incremented by 1. The statement therefore prints 3, but after the statement is executed, the value of *a* is 4. This shows the last *println()* statement. The -- operator works similar but decrements the value of a variable by 1.

With regard of negation you can consider the statements:

```
int a = 2;
int b = 3;
System.out.println(a == b);
System.out.println(!(a == b));
```

If the statements executes you get

```
false
true
```

a == b is a condition, and the first *System.out.println()* statement prints the value of this condition which is *false*. The next statement prints the negation of the condition that is *true*. You should note the parentheses in the condition *!(a == b)*. They are necessary because ! has higher priority than ==.

OPERATORS TO CONDITIONS

It includes the relational operators, comparison operators and the logical operators. Generally, these operators are not the big challenges, but you should be aware of the logical operators. Consider the following statements

```
int a = 2;
int b = 3;
int c = 2;
System.out.println(a == b || a == c);
System.out.println(a == b && a == c);
System.out.println(a == b || a == c && a == c);
```

that prints

```
true
false
true
```

It is not so very strange, but note the last, where && operator is performed before || because of priorities.

THE QUESTIONS OPERATOR

As the last operator I will mention the questions operator, which is the only operator that takes three arguments. It is of the form

condition ? arg1 : arg2;

Here are *arg1* and *arg2* expressions of the same type and the result of the operator is *arg1*, if the *condition* is *true*, and otherwise the value is *arg2*. Consider the following statements:

```
int a = 23;
int b = 17;
System.out.println(a < b ? a : b);
```

They prints

```
17
```

First the *println()* statement evaluate the condition

```
a < b
```

Because it is *false*, the statement prints the value of *b,* that is the smaller of the two numbers *a* and *b*.

EXERCISE 4

Try to determine what the following statements prints – with paper and pencil, and without writing a similar program:

```
int a = 2;
int b = 3;
System.out.println(a + b / 2 == 2);
System.out.println((a + b) / 2 == 2);
System.out.println(a / b);
System.out.println(b / a);
System.out.println(a % b);
System.out.println(b % a);
System.out.println(a > b);
System.out.println(a != b);
System.out.println(a < b ? "Gudrun" : "Olga");
System.out.println(a > b ? "Gudrun" : "Olga");
System.out.println(b == a++);
```

```
System.out.println(--b == --a);
System.out.println(a == b++ && b == 3);
System.out.println(++a == b++ && a < b && a % 2 > 0);
```

When you are finished, you can check the result by writing a program that has a *main()* method with the above statements.

EXERCISE 5

Write a program where the user must enter two integers. The program should then print

- the sum of the numbers
- the numbers difference
- the numbers product
- the numbers qoutient
- the numbers modulus

The result could, for instance be:

```
Enter the 1. number: 123
Enter the 2. number: 13
Sum:         136
Difference:  110
Product:     1599
Qoutient:    9
Modulus:     6
```

4.2 LITERALS

Primitive types are built into the language, and variables of primitive types can be assigned a value using literals, for example

```
boolean res = true;
char ch = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
String s = "Hello World";
```

where the last example show a literal, but not for a primitive type.

There are some syntax for how to define literals, and below are the most important.

LITERALS FOR INTEGERS

Generally, you specify an integer using the digits 0–9, and thus a number from the 10-number system. An integer is interpreted as *long* if it ends with the letter L or l. Otherwise, the type is *int*. It is recommended that you use a large L, as a small l is hard to distinguish from the digit 1.

You can also specify that the integer should be interpreted in hexadecimal by starting the number with 0x, and the symbols are the digits and the letters A, B, C, D, E and F. Finally, you can specify that the integer should be interpreted binary starting the integer with 0b and then the symbols 0 and 1.

Examples could be the following, where the first statement define a literale as a *long* and the last three all define an *int* with the value 26:

```
long t = 123456789012L;
int a = 26;
int b = 0x1a;
int c = 0b11010;
```

LITERALS DEFINING FLOATING POINTS

A floating point or decimal number is interpreted as the type *float* if it ends with the letter F or f. Otherwise, its type is *double*, which can also be set explicitly with the letter D or d. You can also specify an exponent with the letter E or e. The following variables have all the value 1234.56:

```
double x = 1234.56;
double y = 1.23456E3;
float z = 1234.56F;
```

Here you need to specifically noticed that 1.23456E3 means 1.23456 times 10 in the third.

LITERALS DEFINING CHARACTERS AND STRINGS

Literals defining the types *char* and *String* can contain any Unicode (UTF-16) characters, and to the extent that the characters are present, they can be used directly. Is a character is not on your keyboard, it can be added as *Unicode escape sequences* like '\u0108'. For *char* literals use single quotes, while for *String* literals use double quotes:

```
char c = 't';
String s = "Gorm den Gamle";
```

Java also supports single escape sequences for special control characters:

- \b backspace
- \t tabulator
- \n line feed
- \f form feed
- \r carriage return
- \" double quite
- \ single quite
- \\ backslash

Consider as an example the following statements:

```
System.out.println("1234567890");
System.out.println('\u0108');
System.out.println('\'');
System.out.println("abc\tde\\\"fg");
System.out.println("a\nb\nc");
System.out.println("\u0041\u0042\u0043");
```

If done, you get the result:

```
1234567890
Ĉ
'
abc    de\"fg
a
b
c
ABC
```

USE OF UNDERSCORES IN NUMBER LITERALS

As a final note regarding literals, I will mention using '_' in literals for numbers. As examples it is legal to write

```
long kortId = 1234_5678_9012_3456L;
long cprnr = 999999_9999L;
float pi = 3.14_15F;
long ip = 0xFF_EC_DE_5E;
long ord = 0xCAFE_BABE;
long max = 0x7fff_ffff_ffff_ffffL;
byte val = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

and all instances of '_' are ignored. The aim is only to enhance the readability of large numbers. There are a few limitations:

- A number must not start or end with the character _
- The character _ must not stand next to the decimal point in a floating point number
- The character _ must not stand before an F or L

4.3 OBJECTS

Java is an object oriented programming language, and there are some basic concepts you need to know, before you can write programs that perform something interesting, and the following is an overall introduction to objects, classes, inheritance, interfaces and packages, where I primarily will focus on how these concepts relate to the real world, while providing a basic introduction to the syntax of Java.

An object is a software module having a state and a behavior. Software objects are often used to model real-world objects. This section explains how an object representing state and behavior and introduces the concept of data encapsulation. The important thing is to explain the advantages of designing software in this way as a family of cooperating objects.

Objects are the key to understanding the object-oriented technology. If you look around, you will find many examples of real-world objects. It could be the same as a desk, your wife, a particular colleague, a customer, your television, your car and so on. These objects are characterized by two things: They all have state and behavior. A customer has a name, an address, a telefonnunber, a balance. It is the customer's state. The customer can order a product, the customer can return a product, the customer can pay for a product. It is the customer's behavior. In the same way a car's state may be make, color, power, etc., and its behavior might be starting, changing gears, braking and so on. By the way to identify the state and behavior of objects in the real world, you have a good starting to think object-oriented, and thus to find the objects in the development of a program.

Software objects are conceptual, but looks like the real world's objects. They also have a state and related behaviors. An object stores its state in variables or fields, and it performs its behavior through methods. Methods acting on an object's internal state and serves as the primary mechanism for object-to-object communication. The hiding of the objects internal state and require that all interaction must be made using the objects methods is called for data encapsulation and is a fundamental principle of object-oriented programming. With data encapsulation it is the object, which has control over how its state changes as it is the object that through its methods determines what the outside world can do with it.

Above I have dealt with variables in Java, but variables can be several things, and there are basically the following opportunities:

1. Instance variables (non-static fields) as objects use to store their state. All variables that in a class are defined as non-static, are instance variables, and each object of the class has its own instance variables, as a specific object's state is independent of all other objects state, even if it is objects of the same class.
2. Class variables (static fields) are variables defined *static*. Aside from that they are created in the same way as instance variables, but there are only created one copy of these variables, and all objects of a certain class will share all class variables. A static variable is thus designed to store values to be used by all objects of a class.
3. Local variables are in principle created in the same way as instance variables, but they are created in an method, and is known only to this method. All variables that I've used above, are local variables. They are typically used to store temporary values that a method needs to do its job. Instance variables and class variables have a default value if they are not initialized by the programmer. In contrast, local variables must be initialized with a value. A local variable is created when the method in which it is defined, is called, and the variable is automatically deleted again when the method is completed.

4. Parameters are variables that you specify in parentheses after the name of a method. Parameters seems a bit like local variables, but they are used to pass values to a method when it is called. Thus, one can think of parameters as variables that defines the values that a method should act on. *main()* has a parameter called *args* that are used to transfer arguments to the program on the command line.

To construct program code as a family of software objects provides a number of advantages:

- Modularity, where the code for an object can be written and maintained independently of the code to other objects.
- Information hiding, where all interaction with an object is by means of its methods. The details concerning state of the object and the internal implementation remain hidden from the outside world.
- Code Reuse, where already developed objects (perhaps written by another software developer) can be used in the current program. It allows the objects to be developed and tested by specialists, which you can trust on and use them in your own programs.
- Modifiable, where a particular object without affecting the rest of the program can be replaced with another, if the object for some reason is found inexpedient.

In the real world, you often meet many individual objects that are similar and of the same kind. There are thousands of cars, all of the same make and model and is constructed in exactly the same way and of identical components. In the object-oriented world they wee say that a particular car is an instance of the class cars of a particular make and model. A class is a description of how specific cars behave.

Wee use the same principle in relation to software. Assume that a program need objects for banknotes. A banknote is characterized by a value which is the banknote's state, and the only thing you have to do with a banknote is to read its value and know how the note looks so you can distinguish it from other notes. Accordingly, you can define the concept of a banknote as follows:

```
package bankprogram;

public class BankNote
{
 private int value;

 public BankNote(int value)
 {
  this.value = value;
 }

 public int getValue()
 {
  return value;
 }
}
```

It is an example of a Java *class*, which here has the name *BankNote*. A definite note has as mentioned a value that is the note's state, and this value must be stored somewhere. To this end, the class defines a variable called *value*. A variable has a type, here an *int* and means that the variable may contain an integer. The variable *value* is an example of an instance variable. The variable is defined *private*, which means that it can only be referenced from the class itself, and that is what I have called data encapsulation. The class should be used for something, and you should be able to read the note's value. The class has a method named *getValue()*, whose value is the banknote's value that is the value of the variable *value*. This method defines the objects behavior. Note that the method has a type, which here is *int*. Wee say that the method returns a value. The class has another method that has the same name as the class. It is a special method, which does not define the behavior of an object, but is used to initialize an object's state when creating a new object. Such a method is called a *constructor*.

You should note that the class has no *main()* method, because the class is not a program, but merely defines a concept that can be used by any application.

With the class *BankNote* available, a program can create *BankNote* objects that it can do something with. Below is a *main()* method from a program that creates two *BankNote* objects. The first has the value of 100, while the next has the value of 200:

```
package bankprogram;

public class BankProgram
{
 public static void main(String[] args)
 {
 BankNote note1 = new BankNote(100);
 BankNote note2 = new BankNote(200);
 System.out.println(note1.getValue());
 System.out.println(note2.getValue());
 }
}
```

You must specifically note how to create an object. An object has a name, for example *note1*, which is a variable. The object also has a type, which is here *BankNote*, and when the type is a *class* rather than a primitive type, the object must be created with the *new* operator. Here you specify the value of the banknote, and exactly it means that the class's constructor is executed and transmits the value to the object's instance variable *value*. If the *main()* method is performed it prints the values of the two *BankNote* objects. For that to be possible, you have to refer to the *BankNote* object's value. It is not possible because of data encapsulation, but a *BankNote* object has a behavior in terms of the method *getValue()*, so you can refer to the value.

INHERITANCE

Different kinds of objects often have several characteristics in common with each other. If you think of banknotes they all have a value, but they differ with respect to how they look. They have different colors, and they show different pictures. In object-oriented programming it is possible that classes can inherit each other. That means that you can collect common properties in a base class, while the differences can be placed in derived classes. Below is a class that represent a note and has the value 100 (a danish banknote):

```
package bankprogram;

public class BankNote100 extends BankNote
{
 public BankNote100()
 {
  super(100);
 }

 public void print()
 {
  System.out.println("Den gamle Lillebæltsbro og Hindsgavl-dolken");
 }
}
```

The class is called *BankNote100*, and after the class name you defines with the word *extends* that the class inherits the class *BankNote*. This means that the class inherits all the properties that a *BankNote* have and expanded with new properties. In this case *BankNote100* extends *BankNote* with a method *print()*, to simulate how the note looks. The class *BankNote* has a constructor which initialize the instance variable *value*. A *BankNote100* must also initialize this variable (the value 100). It happens in the class's constructor with *super(100)*, which means that the base class's constructor is executed.

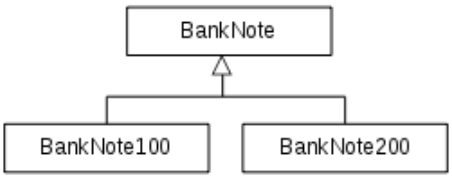
It is clear that in quite the same way you write a class *BankNote200* representing a banknote with the value 200.

With these classes available you can write the following *main()* method:

```
public static void main(String[] args)
{
 BankNote100 note1 = new BankNote100();
 BankNote200 note2 = new BankNote200();
 note1.print();
 note2.print();
 System.out.println(note1.getValue() + note2.getValue());
}
```

Here, you create two objects of the types respectively *BankNote100* and *BankNote200*. Next the objects *print()* method is called. Note the syntax, and how to perform a method on an object using the dot operator. The last statement prints the sum of the two objects values. Here you should note the use of the method *getValue()*. It is defined in the class *BankNote*, but is inherited to the classes *BankNote100* and *BankNote200*.

Often wee illustrate inheritance as follows:



In this context the base class *BankNote* is also called a *superclass* while the derived classes *BankNote100* and *BankNote200* are called *subclasses*. Sometimes wee also say that *BankNote* is a generalization of *BankNote100* and *BankNote200* while these classes are called specializations of *BankNote*.

INTERFACES

An object's properties are known to the outside world through the *public* methods that the class defines and thus made available as services. Methods form the object's interface to the outside world. In order to define an object's characteristics, you can use an interface which defines the methods of an object of a certain class. For example you can define a banknote as follows:

```
package bankprogram;

public interface Note
{
 public int getValue();
 public void print();
}
```

It looks like the definition of a class, but you should notice that it simply is only a definition and that the interface does not contain code that implements something.

Given an interface you can define that a class must implement this interface. Thus you specify that the class has the methods that the interface defines. For example the class *BankNote* can implement the interface *Note* as follows:

```
package bankprogram;

public abstract class BankNote implements Note
{
 private int value;

 public BankNote(int value)
 {
  this.value = value;
 }

 public int getValue()
 {
  return value;
 }
}
```

Note first the word *implements* that defines that the class implements an interface. This means that the class should have two methods *getValue()* and *print()* – it must implement the methods that the interface defines. It is not the case, since it do not implements the method *print()*. It can not do that because it does not know how this method should be written. It is the derived classes *BankNote100* and *BankNote200*, that has that knowledge. The problem is solved by defining the class as *abstract*. The method then has to be implemented in the derived classes, what it indeed is. That a class is *abstract* has further that consequence that you can not instantiate objects whose type is the abstract class – in this case *BankNote*. It is also reasonable, else you could instantiate notes with an arbitrary value, which does not model the real world in a meaningful way. Below is a new version of *main()* program:

```
public static void main(String[] args)
{
 Note note1 = new BankNote100();
 Note note2 = new BankNote200();
 note1.print();
 note2.print();
 System.out.println(note1.getValue() + note2.getValue());
}
```

There will again be created two objects, respectively a *BankNote100* and a *BankNote200* object, but their type is *Note*. A *BankNote100* is especially a *BankNote*, which is again a *Note*. This means that the two objects in the rest of the program alone is known from the defining interface and then in the rest of the program are objects that provides two services *getValue()* and *print()*.

From the above, it is not obvious what the advantage of interfaces are and it will first be clear later, but the explanation must be sought in that, *note1* and *note2* in the program only are known from the defining interface, which means that the two objects are treated equally independent their specific types.

The project *BankProgram* implements the above classes and interface.

PACKAGES

A package is a so-called namespace that organizes a family of related classes and interfaces. Conceptually, you can think of a package in the same way as a folder in which a package contains classes which somehow belong together. Since programs written in Java may consist of hundreds or thousands of individual classes, it makes sense to keep things separated by placing related classes and interfaces in packages.

When NetBeans create a project, it also create a package and place the project's classes and interfaces in that package, for example

```
package bankprogram;

public interface Note
{
 public int getValue();
 public void print();
}
```

which means that the interface is in the package *bankprogram*. So long that the project contains only a single package, it is not something you need to think about, but it means exactly that the name of the type *Note* is

```
bankprogram.Note
```

A *package* statement defines the package that a type belongs. It must be the first statement in the file that defines a type, but may be preceded by a comment.

The Java platform includes a huge library (a collection of packages), which contains components, the programmer may use in his own applications. This library is known as the *Application Programming Interface* usually abbreviated to *API*. Its packages represents the tasks most often associated with general issues within the program development, and examples are packages for developing applications with a graphical user interface.

To refer to classes other than in the defining package, the package must be referenced with an *import* statement. The following statement

```
System.out.print("100 kr., ");
```

prints a text, and *System* is actually the name of a class that exists in the package *java.lang*. This package contains all of the most common classes from the Java API, and when it is not necessary to an *import* statement for this package, it is because the compiler as default imports the package.

EXERCISE 6

Create a copy of the project *BankProgram*. In the class *BankNote100*, you should modify the *print()* method as follows:

```
public void print()
{
 System.out.print("100 kr., ");
 System.out.println("Den gamle Lillebæltsbro og Hindsgavl-dolken");
}
```

Test the program and note that the text of both statements is on the same line.

Modify the *print()* method in the class *BankNote200* in the same way.

Write three classes *BankNote50*, *BankNote500* and *BankNote1000* representing banknotes with values respectively 50, 500 and 1000. The three classes are of course similar to the classes *BankNote100* and *BankNote200,* but the value is not the same and the text in the *print()* is different. The classes represents Danish banknotes, and the notes text are respectively

1. Sallingsundbroen og Skarpsalling-karret
2. Dronning Alexandrines bro og bronzespanden fra Keldby
3. Storebæltsbroen og Solvognen

In *main()*, create an object of each of the five specific banknote types. You should print the five objects and the sum of the five banknote's values.

PROBLEM 3

You should write a program that creates objects that represents employees in a company. You should solve the problem by following the steps below.

1)

Create a new project in NetBeans, that you can call *Employees*. Add the following interface to the project:

```
package employees;

// Interface that defines an employee of a company
public interface Employee
{
```

```
 public String getName();      // returns the employee's name
 public int getSalary();       // returns the employee's monthly salary
 public void print();          // prints information about the employee
}
```

This interface defines an employee and in this exercise an employee do not have any other characteristics.

2)

Add an abstract class *AbstractEmployee*, which implements the interface *Employee* and represents an employee when

1. the class should have two variables, for respectively the name and the monthly salary
2. the class should have a constructor that initializes the two variables
3. the class should implement the two methods *getName()* and *getSalary()*

3)

Add a class *Bookkeeper* who inherit *AbstractEmployee*. Besides the constructor (which must have two arguments) the class must implement the method *print()*, that the prints the name, the job title and monthly salary. The result could, for instance be:

```
########################################
Gudrun Jensen, Bookkeeper
Monthly salary: 45000
########################################
```

where the name and monthly salary are the values passed to the constructor.

4)

Add a corresponding class *Janitor* that represents a janitor. This class is basically identical to the above but the *print()* method should be implemented slightly differently, so the result could be:

```
========================================
Karlo Hansen, Janitor
Monthly salary: 35000
========================================
```

5)

Finally, write a class *Director*, representing a director. A director's monthly salary will consist of the agreed monthly salary and a monthly bonus to be passed as an argument to the constructor. The class must therefore have a variable to this value:

```
public class Director extends AbstractEmployee
{
 private int bonus; // monthly bonus
```

The class must also implement the method *getSalary()*, even though it is implemented in the base class, as it must now operate in a different way. Its value shall consist of the value of the base class method, and the value of the variable *bonus*, and the method can be written as follows:

```
public int getSalary()
{
 return super.getSalary() + bonus;
}
```

You should just accept the syntax, but the word *super* means that you refers to the method *getSalary()* in the base class.

The *print()* method must be rewritten so that the result for example could be the following, but the value of the monthly salary shall be without bonus:

```
****************************************
****************************************
Abelone Andersen, Dircktor
Monthly salary: 55000
****************************************
****************************************
```

6)

Write finally a program – a *main()* method – which does the following:

1. Creates an object of the type *Employee*, that is a *Bookkeeper* named *Gudrun Jensen* and with a monthly salary that is 45000
2. Creates an object of the type *Employee*, which is a *Janitor* named *Karlo Hansen* and with a monthly salary that is 35000
3. Creates an object of the type *Employee*, which is a *Director* named *Abelone Andersen* and with a monthly salary that is 55000 and a bonus on 15000
4. Prints the three objects
5. Prints the sum of the three employees monthly salary

If you executes the program, the result should be as shown below:

```
########################################
Gudrun Jensen, Bookkeeper
Monthly salary: 45000
########################################
========================================
Karlo Hansen, Janitor
Monthly salary: 35000
========================================
****************************************
****************************************
Abelone Andersen, Director
Monthly salary: 55000
****************************************
****************************************
150000
```

4.4 EXAMPLE: CUBES

I will show a class representing a usual cube with six sides. The class can be written as follows:

```
package cubes;

import java.util.*;
// Class that represents a normal cube
public class Cube
{
 private static Random rand = new Random();
 private int eyes;

 public Cube()
 {
  roll();
 }

 public int getEyes()
 {
  return eyes;
 }
```

```
  public void roll()
 {
  eyes = rand.nextInt(6) + 1;
 }

 public String toString()
 {
  return "" + eyes;
 }
}
```

The class has two fields: A class variable and an instance variable. The class variable is called *rand* and is an object of the type *Random*. It is a random number generator which can generate random numbers. It works by reading the machine clock and then, produces a sequence of values, based on the clock's value when the object is created. As you can not know (assume) something about what the clock shows at any given time, the start time is randomly and the sequence of generated numbers appear to be random. The variable *rand* must be a class variable, when objects of type *Cube* otherwise would have their own random number generator, which could easily result in being initialized with the same value of the clock. The result would be that a number of *Cube* objects would create the same sequence of random numbers. The class's instance variable is called *eyes* and is used to keep track of the value of each *Cube*. Since two *Cube* objects do not necessarily have the same value, this variable must be an instance variable, so each object has its own copy.

The class has a method called *roll()*, and simulating that you throw the cube. When the method is performed the class variable *rand* is used. The class *Random* has a method called *nextInt()*, which returns a random non-negative integer. In this case, it is one of the values 0, 1, 2, 3, 4 and 5, and when one adds 1 the result is a random value between 1 and 6 both inclusive. The class *Random* is defined in a package called *java.util*. In order to use classes in this package, it is necessary with an *import* statement. Here the * means that you can use all classes in the package.

The class *Cube* also has a constructor. Since the variable *eyes* is not initialized, the default value is 0. This is unfortunate because it is an illegal value. Therefore the constructor rolls the cube and thus ensures that the cube's value from the start is legal.

The class has two other methods. The method *getEyes()* is a method that returns what the cube shows. It looks like what you met above. Then there are the method *toString()*, which returns the value of the cube as a text. The syntax of the return statement is not obvious, but it force the method to return the value of the variable *eyes* as a text.

The class could be used in a *main()* method as follows:

```
public static void main(String[] args)
{
 Cube c1 = new Cube();
 Cube c2 = new Cube();
 do
 {
  c1.roll();
  c2.roll();
  System.out.println(c1 + " " + c2);
 }
 while (c1.getEyes() != c2.getEyes());
}
```

It requires a little explanation when I uses statements which I have not yet mentioned. The first two statements creates two *Cube* objects, and to what has been said above, there is nothing new in it. The next statement is a *do* statement and is an example of a loop. It means that the three statements in the block between { and } is performed until the condition after *while* is *false*. The condition uses the operator != meaning different from, and thus the condition is *true*, as long as the two *Cube* objects shows a different number for the eyes. The block with the three statements rolls the cubes and prints them. Here you should note that each iteration of the loop prints

```
c1 + " " + c2
```

and here it is the value of the *Cube* class's *toString()* method that are printed. The result is that the program rolls the two cubes and prints them until they shows the same number of eyes.

A *do* statement is an example of a statement for program control and are discussed in the next chapter, but when you see the code, it is not particularly difficult to find out what happens.

EXERCISE 7

In this exercise, you should write a program, similar to the above, but instead of cubes the program should works with coins.

Start with a new NetBeans project, you can call Coins.

Add a class called *Coin* that represent a coin that can show *head* or *tail*, that is a coin that can flip. The class should look likes the class *Cube* and should have a random number generator. In addition, it must have an instance variable to the coin's value:

```
private char value;
```

The value must be *H* or *T*. The class must include a constructor and should have three methods:

- *getValue()*
- *toString()*
- *void roll()*

Here, the first two are written in the same way as in the class *Cube*, but the latter requires a little more, and can be written as follows using the questions operator:

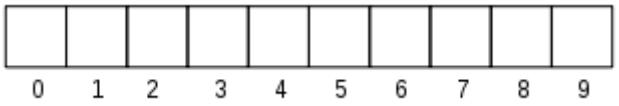
```
public void roll()
{
 value = rand.nextBoolean() ? 'H' : 'T';
}
```

Here, the random number generator is used to return a random *boolean* value, and depending on this value the method assigns the variable *value* the value '*H*' or '*T*'.

You must then write a *main()* method that creates two *Coin* objects and then uses a *do* loop to throw the coins and print the two coins until they have the same value.

4.5 ARRAYS

An array is a container for objects of a certain type, and an array is itself an object. An array can contains a certain number of elements, and the number of elements an array can contains are defined when the array is created. One may illustrate an array as follows



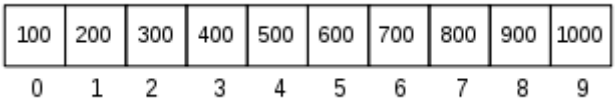
illustrating an array with space for 10 elements. Each element is identified by an index, starting with 0. Consider as an example the following *main()* method:

```
package array01;

public class Array01
{
 public static void main(String[] args)
 {
  int[] arr = new int[10];
  arr[0] = 100;
  arr[1] = 200;
  arr[2] = 300;
  arr[3] = 400;
  arr[4] = 500;
  arr[5] = 600;
  arr[6] = 700;
  arr[7] = 800;
  arr[8] = 900;
  arr[9] = 1000;
  System.out.println("Element at index 0: " + arr[0]);
  System.out.println("Element at index 1: " + arr[1]);
  System.out.println("Element at index 2: " + arr[2]);
  System.out.println("Element at index 3: " + arr[3]);
  System.out.println("Element at index 4: " + arr[4]);
  System.out.println("Element at index 5: " + arr[5]);
  System.out.println("Element at index 6: " + arr[6]);
  System.out.println("Element at index 7: " + arr[7]);
```

```
  System.out.println("Element at index 8: " + arr[8]);
  System.out.println("Element at index 9: " + arr[9]);
 }
}
```

Here you creates an array named *arr*, which has space for 10 elements of the type *int*. After the array is created, all elements has the value 0. The next 10 statements assigns values to all places in the array:



If the *main()* method is performed, you get the result:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

The type of an array can be anything, including a class type. As an example could the above program *Cubes* be written as follows:

```
package array02;

public class Array02
{
 public static void main(String[] args)
 {
  Cube[] c = new Cube[2];
  c[0] = new Cube();
  c[1] = new Cube();
  do
  {
   c[0].roll();
   c[1].roll();
   System.out.println(c[0] + " " + c[1]);
  }
```

```
  while (c[0].getEyes() != c[1].getEyes());
 }
}
```

where the *main()* method creates an array with space for two *Cube* objects. When the type of an array is a class type, however, there is an important difference, since the individual elements must be created explicitly. When writing

```
Cube[] c = new Cube[2];
```

main() creates an array with space for two *Cube* objects, but the two places are empty, which means that they have the value *null*, that indicates that the place is empty (it does not refer to anything). The individual objects has to be created explicitly as shown in the following two statements in the program.

You can also let the compiler create an array from a list of values:

```
int[] arr = { 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 };
System.out.println("Element at index 0: " + arr[0]);
System.out.println("Element at index 1: " + arr[1]);
System.out.println("Element at index 2: " + arr[2]);
```

Here the compiler will create the array with the length that the number of elements in the list indicates and the elements in the array are initialized with the values in the list.

For the array of *Cube* objects, one could write

```
Cube[] c = { new Cube(), new Cube() };
do
{
 c[0].roll();
 c[1].roll();
 System.out.println(c[0] + " " + c[1]);
}
while (c[0].getEyes() != c[1].getEyes());
```

The following program creates an array with 5 integers of the type *int* and initializes the elements with random numbers, then the numbers are printed on the screen:

```
package array05;

import java.util.*;

public class Array05
{
 private static Random rand = new Random();

 public static void main(String[] args)
 {
  int[] arr = new int[5];
  for (int i = 0; i < arr.length; ++i) arr[i] = rand.nextInt();
  for (int i = 0; i < arr.length; ++i) System.out.println(arr[i]);
 }
}
```

The program has a random number generator and creates an array with space for 5 elements of the type *int*. Next, the array is filled with random integers. This is done with a *for* loop, which is a control structure that I have not yet mentioned, but it is easy enough to understand how it works. It has three parts. In the first part defines a variable that is initialized to 0. The second part is a condition that tests whether the value of the variable *i* is less than the number of elements in the array. You must specifically noting how one refers to the number of elements in an array as

```
arr.length
```

If the condition is true, the statement after the *for* statement is executed, which here is

```
arr[i] = rand.nextInt();
```

This means that the *i*-th element in the array is assigned a value. Next, the third part of the *for* statement is executed that add 1 to the variable *i*, and then the statement test the condition again and everything repeats itself. Since the variable *i* for each iteration is 1 higher, all places in the array are referenced until there are no more. *main()* has another *for* statement, and it works in principle the same way. It runs through all the array's elements place for place. The first *for* statement initialize the elements, while the other prints them.

As another example, the following program defines an array with three elements of the type *String* and prints them:

```
package array06;

public class Array06
{
 public static void main(String[] args)
 {
  String[] arr = { "Svend", "Knud", "Valdemar" };
  for (int i = 0; i < arr.length; ++i) System.out.println(arr[i]);
 }
}
```

EXERCISE 8

Create a new project in NetBeans, that you can call *ArrayDemo*.

Add a static method *test1()* that creates an array with space for 10 elements of the type *double*. The method should then fill the array with random numbers of the type *double*, and finally print the content of the array. Call the method from *main()* to test it.

Add another method *test2()*, that from a list creates an array of the type *char* with the first 5 capital letters. The method should then print the array. Test the method from *main()*.

Add a method *test3()* that create an array with space for 10 elements of the type *boolean*. The method must then fill the array with random values and finally print the content of the array.

Write a method that has an array as a parameter

```
static void print(int[] arr)
{
}
```

The method should print the contents of the array, so all elements are on the same line, separated by a space.

Write a last method *test4()*, that creates an array with space for 10 elements of type *int*. The method must fill the array with random 2-digit integers, and finally the method must print the array by using the above *print()* method.

4.6 EXAMPLE: *CUPPROGRAM*

In section 4.4, I have shown a class *Cube* representing a usual cube with six sides. In this section I will show a class that can represent a cube cup. It is a container, which may include cubes, and therefore the class internal can be implemented as an array of the type *Cube*. The class must have a method that can simulate that you toss with the cup, and also there must be a method that displays the content of the cup. Correspondingly, the class can be written as follows:

```
package cupprogram;

// Class representing a cup with cubes.
public class Cup
{
 private Cube[] arr;

  public Cup(int n)
 {
  arr = new Cube[n];
  for (int i = 0; i < n; ++i) arr[i] = new Cube();
 }

  public void toss()
 {
  for (int i = 0; i < arr.length; ++i) arr[i].roll();
 }

  public boolean yatzy()
 {
  for (int i = 1; i < arr.length; ++i)
   if (arr[i].getEyes() != arr[0].getEyes()) return false;
  return true;
 }

  public String toString()
 {
  String text = arr[0].toString();
  for (int i = 1; i < arr.length; ++i) text += " " + arr[i];
  return text;
 }
}
```

The variable *arr* is an array of type *Cube*, but it is not created in the declaration but in the constructor. The reason is to make the class more general and the constructor has a parameter that tells how many *Cube* objects the cup must contain. You should note that the constructor explicitly must create the concrete *Cube* objects that happens in a *for* loop. It is necessary, since the statement

```
arr = new Cube[n];
```

simply creates an array with space for *n Cube* objects, but not the concrete *Cube* objects. The method *toss()* simulates that you toss with the cup. This is done by using a *for* loop that traverses all the *Cube* objects in the array *arr* and throwing them by calling the metod *roll()* for every *Cube* object. Instead of a method that prints the contents of the cup, the class has a *toString()* method that returns the contents of the cup as a *String*, that is the values of the individual cubes separated by spaces.

The class has a method called *yatzy()* which returns *true* or *false*. The method must test if all cubes are the same, and it is done by a *for* loop that iterate through the cubes and by means of an *if* statement test whether there is a cube, which is not equal to the first cube (has the same value as the value of the first cube). In this case, all the cubes are not identical, and the method returns *false*. Coming throughout the loop, it means that all cubes are equal to the first cube, and thus all cubes are equal, and the method returns *true*.

if is also an example of a control structure, and it will also be explained in the next chapter, but it allows to introduce a condition in a program so that a statement is executed only if the condition is *true*. In this case, the statement is

```
if (arr[i].getEyes() != arr[0].getEyes()) return false;
```

and after *if* there is a condition in parentheses. If the condition is *true*, this means that the two cubes have a different number of eyes, and the next statement is performed, which is a return statement. This means that the method ends with the value *false*. If the condition is not *true*, the subsequent statement is simply ignored, and in this case it means that the *for* statement continues.

The method *yatzy()* is included for the following example that the use the class *Cup*, but really the class ought not have such a method, and verifying that all cubes are the same, should be solved in a different way. The problem is that classes must reflect concepts from the real world, and in practice you can not ask a cup, where all cubes are the same, but you must manually inspect each cube. A method as *yatzy()* makes the class *Cup* more specialized and it is not as easy to use in other contexts.

As an example of the use of the class *Cup* is shown a program that creates a cup with 5 cubes. Then the program simulates a game, where to toss with the cup until all cubes are the same:

```
package cupprogram;

public class CupProgram
{
 public static void main(String[] args)
 {
  Cup cup = new Cup(5);
  do
  {
   cup.toss();
   System.out.println(cup);
  }
  while (!cup.yatzy());
 }
}
```

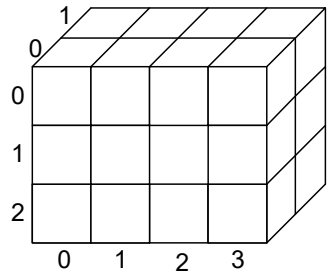
4.7 MULTIDIMENSIONAL ARRAYS

An array as described above is sometimes called a one-dimensional array corresponding to that it is a number or sequence of elements in which the elements are identified by a single index. You can also organize the elements in a table:

2	3	5	7
11	13	17	19
23	29	31	37

and here the individual elements are identified by an index pair consisting of a row index and a column index. For example is the element 17 identified as the element with index (1, 2), that is the element in row 1 and column 2. One talks in this case of a 2-dimensional array.

You can continue with more dimensions and in principle all the dimensions, as may be needed, and a cube with 3 dimensions is a 3-dimensional array, where the individual elements are identified by a triple (i, j, k). For arrays of dimension greater than 3 wee have no immediate visual representation, but in programming you can work with arrays of arbitrary high dimension. In practice, however, it is rare to use arrays with dimensions greater than 2.



Java does not directly support multidimensional arrays, but since an array can be of any type, the type can particularly be another array. Therefore, one can implement a 2-dimensional array as a array of arrays. Consider the following method:

```
static void fill(int[][] arr)
{
 for (int i = 0; i < arr.length; ++i)
  for (int j = 0; j < arr[i].length; ++j) arr[i][j] = rand.nextInt(10);
}
```

The method has a parameter called *arr*, whose type is *int[][]*. An array is defined as

```
type[] array;
```

and *arr* is an array of type *int[]*, that is an array whose elements are arrays of the type *int*. The first *for* loop in the method *fill()* is a standard iteration of an array in which each element *arr[i]* is an array with elements of the type *int*. The second *for* loop performs for each of these arrays an iteration where it assigns each element a random integer between 0 and 9, both included. The result is that the *fill()* method initialize a 2-dimensional array with random integers.

Consider then the following *print()* method, which prints a two-dimensional array. This is done by using the *print()* method for a 1-dimensional array from exercise 8:

```
static void print(int[][] arr)
{
 for (int i = 0; i < arr.length; ++i) print(arr[i]);
}
```

The parameter *arr* is a 2-dimensional array, and each element in *arr* is therefore an usual 1-dimensional array. The *for* loop sends each element in *arr* to the first *print()* method, that prints the element (which is a 1-dimensional array) as a single line, where the numbers are separated by spaces.

Below is an example of how to create a 2-dimensional array with 10 rows and 5 columns:

```
int[][] arr1 = new int[10][5];
```

You can also create a two-dimensional array from a list:

```
int[][] arr = { { 2, 3, 5, 7 }, { 11, 13, 17, 19 }, { 23, 29, 31, 37 } };
```

which creates an array with three rows and four columns. The rows do not have to be of the same length, and such the following statements create an array with 10 rows, each row having a random length of between 1 and 10:

```
int[][] arr = new int[10][];
for (int i = 0; i < arr.length; ++i) arr[i] = new int[rand.nextInt(10) + 1];
```

EXERCISE 9

This exercise is a continuation of exercise 8. Make a copy of the solution to exercise 8 and open the copy in NetBeans.

Implements (that is write the code for) the two methods *fill()* and *print()* above. Compile the project so you are sure that there are no syntax errors.

Write a method *test5()*, which creates a 2-dimensional array with elements of the type *int*, which has 10 rows and 5 columns. Next, fill the array using the method *fill()* and print it using the method *print()*. Test *test5()* from *main()*.

Write a method *test6()*, which creates a two-dimensional array with four rows and five columns. The array should be created from a list of the first 20 primes. The method must also print the array using the method *print()*.

Write a method *test7()*, which creates a 2-dimensional array with elements of type int. The array must have 5 rows, and the length of the columns should be 3, 5, 1, 7 and 5. Next, fill the array using the method *fill()* and print it using the method *print()*.

5 PROGRAM CONTROL

In this chapter I will deal with control statements in Java, and in fact I have already used three of them:

- *if*
- *do*
- *for*

The first is used to write conditions in the program, while the last two is for loops. There are two other control statements, and with these statements available, one is able to write programs that perform something concrete and real data processing, and not just programs that prints text on the screen. Therefore, this chapter will largely consist of examples so you are presented for the development of Java applications in practice. Put differently, this chapter focuses on algorithms.

When wee talk about control statements, it is because it's statements, which controls the execution of other statements which partly can be a single statement and also can be a block. A block encapsulates statements using the parentheses { and } and a block of statements can be treated as a whole, for example using control statements. Blocks appear in several places in Java, for example are the statements in a method a block.

General, statements are performed in the order as they are written in a method, but using control statements, it is possible to deviate from this sequential order, which is necessary in general to be able to write programs that solve something interesting.

5.1 THE *IF* STATEMENT

Java has 9 control statements, and I have already mentioned and used *if*. As an example you can write something like the following:

```
if (a > b) a -= b;
```

After the word *if* is a condition in parentheses, and the meaning is that the subsequent statement that here is an assignment, only is performed if the condition is *true*. If not, the statement is ignored. As another example, the following code shows an *if* statement that controls a block:

```
if (a > b)
{
 int c = a;
 a = b;
 b = c;
}
```

The block consist of three statements. In general, a block may consist of any number of statements and especially declaration of a variable. In particular, attention should be paid to the fact that if a variable is declared in a block, so it is only known within the block. It is local to the block and is created when the program goes into the block where it is defined. It will be removed again when the program exits the block. Note that the above *if* statement reverses the values of the two variables *a* and *b*, when *a* is greater than *b*.

Formally, the syntax of an *if* statement is

if (condition) block

wherein *block* is a block of statements. If the block consists of a single statement, it is allowed to omit block boundaries { and } – as in the first example above. Some choose because of readability always to use { and }, even if there is only one statement, and it is up to you to choose what you find most readable. The condition must be an expression whose type is *boolean*, and thus an expression that evaluates to *false* or *true*.

IF-ELSE STATEMENT

Another control statement is called an *if-else* statement, but it is little more than an extension of the *if* statement. As an example you can write the following:

```
if (a > b) a -= b; else b -= a;
```

The meaning is that if the condition is *true*, then the statement after the parentheses is executed, and otherwise the statement after *else* is executed. The difference is that *if-else* always does something. Formally, the syntax is:

if (condition) block_1 else block_2

The *if* statement and the *if-else* statement are also called for selections (or branches) corresponding to that they make the execution of statements depending on a condition.

In practice, the *if* statement and the *if-else* statement are the most common control statements, and it is hard to imagine a program where these statements do not occur.

EXERCISE 10

Write a program *Sort2* where the user must enter two integers. The program should then print the two numbers in ascending order.

The program should be written as follows:

Create a new project in NetBeans, as you call *Sort2*. You must then add a method to the main class to be used for entering a number:

```
private static int enter()
{
 // create a scanner to entering
 // print a text use the scanner to enter the number
 // convert the value entered to an int and return the number – the converting
```

```
 // must be done in a try/catch and if there is an exception the method should
 // return 0
}
```

You must then write the *main()* method:

```
public static void main(String[] args)
{
 // use the method enter() to enter to numbers
 // if the first number is larger than the other, the numbers must be reversed -
 // see above the if how you do that
 // print the two numbers
}
```

When finished, test program, so you are sure that it does the right thing. The result could, for instance be:

```
Enter a number: 29
Enter a number: 23
23 29
```

PROBLEM 4

In this example, you must write a program, similar to the above, but where the user instead must enter three numbers and the program must then print the three numbers in ascending order.

That there are three numbers instead of two numbers complicates actually the problem a lot, but you can go as follows.

Create a new project, that you can call Sort3.

You can then take a copy of the input method of exercise 10, but the method should be changed so that in the case of an error (the user has entered an illegal numbers) the method prints an error message and stops the program.

In *main()* you can use the method to enter the three numbers a, b and c. You can then sort the three numbers using the following algorithm

```
if a > b then swap the two numbers
if b > c then swap the two numbers
if a > b then swap the two numbers
```

Print the three numbers *a*, *b* and *c*.

Below is an example of an application of the program where I run it from NetBeans:

```
Enter a number: 19
Enter a number: 17
Enter a number: 13
13 17 19
```

PROBLEM 5

In school you learn in mathematics education to solve a quadratic equation, an equation of the form

$$ax^2 + bx + c = 0$$

where *a* not is 0. In order to solve the equation you has to calculate the discriminant

$$d = b^2 - 4ac$$

Now that there are no solutions if d <0, and there is one solution if d = 0, which is determined by the formula

$$x = \frac{-b}{2a}$$

If d> 0, there are two solutions as determined by the following formulas:

$$x_1 = \frac{-b - \sqrt{d}}{2a}, x_2 = \frac{-b + \sqrt{d}}{2a}$$

The task now is to write a program where the user must enter the coefficients *a*, *b* and *c* of a quadratic equation. If one of the coefficients are not a legal decimal number, the program shouldl stop with an error message. The same should happen if *a* is zero.

Below is an example of running the program:

```
The coefficient a: : 2
The coefficient b: : -10
The coefficient c: : 12
The equation has the solutions:
2.0
3.0
```

To do the exercise you need to know how Java determines the square root of a number. This can be done in the following way:

```
Math.sqrt(d)
```

where *d* is a variable. You can make the code more simply by changing the input method from problem 4, so you transfer the text as a parameter:

```
private static double enter(String text)
{
```

Note that the value entered this time should be converted to a *double*, and that the method therefore also need to return a *double*.

5.2 DO AND WHILE STATEMENTS

I have previously used the *do* statement and the syntax is

do
 blok
while (betingelse);

The construct executes the block, after which the condition is evaluated. If it is *true* the block is executed again, and it is repeated until the condition becomes *false*. You should note that the block is always executed at least once.

A do statement is an example of an iteration, or loop, which means that a statement or a block is executed several times. Another loop is called a *while* statement, and it has the syntax

while (betingelse) blok

and also means that the block is executed as long as the condition is *true*. An example would be

```
int sum = 0;
int n = 2;
while (n <= 1000)
{
 sum += n;
 n += 2;
}
```

that determines the sum of all positive even numbers which is less than or equal to 1000.

EXERCISE 11

You must write a program that can be executed from the command line using the following syntax

```
java -jar Sum 999 999
```

where there are two arguments, which are both integers. The program should print the sum of all integers between the two arguments (both included). In the case that there not are two arguments, or the arguments are not legal integers, the program should print an error message.

You can use the following procedure:

Create a new NetBeans project that you should call *Sum*.

In *main()*, write an *if-else* statement that tests whether there are two arguments

```
if (args.length == 2)
{
}
else
{
}
```

In the *else* part the program should print an error message and nothing else. If there are two arguments, they are both strings, and they must be converted to *int* values. It can can raise an exception. This exception should be treated, and for that you need to introduce a try/catch block, for the *if* part:

```
if (args.length == 2)
{
 try
 {
 }
 catch (Exception ex)
 {
 }
}
else
{
}
```

This means that if one of the statements in the try block returns an error, the try block is interrupted, and the control is transferred to the catch block. In this case, the catch block should not do anything except print an error message. The two arguments can be converted into integers as follows:

```
try
{
 int a = Integer.parseInt(args[0]);
 int b = Integer.parseInt(args[1]);
 long sum = 0;
}
```

and note again that in the case of an error (if one of the two arguments is not an integer), then the program jumps to the catch block. Note also that when the type of the variable *sum* is defined as a *long*, this is because that the sum of all the integers in a range can quickly become large.

You must complete the try block so that you use a *while* statement to determines the sum of all integers from *a* to *b* and print the result.

PROBLEM 6

You should write a program where the user must enter a number of integers. The entry shall continue until the user enters the number 0. Then the program must print

- how many numbers are entered
- the sum of the numbers
- the smallest number
- the largest number
- and the average of the numbers entered

An example of an execution of the program could be as shown below:

```
Enter a number: 19
Enter a number: 3
Enter a number: 23
Enter a number: 5
Enter a number: 11
```

```
Enter a number: 13
Enter a number: 2
Enter a number: 17
Enter a number: 29
Enter a number: 7
Enter a number: 0
Number entered numbers:              10
The sum of entered numbers:          129
The smallest nubmer:                 2
The largest number:                  29
The average of the numbers entered:  12.9
```

To enter a number, you can use the input method from problem 4, but it must be changed so that in the case of entering an illegal number it prints an error message, after which the entry must be repeated.

5.3 THE *FOR* STATEMENT

The last statement to iterations is called *for*, and I have already used the statement many times in when I mentioned arrays. It is the most flexible of the three control statements for loops and also the most used. Formally, the syntax is

for (initialisering; betingelse; udtryk) blok

The initialization is performed only once and will typical initialize one or more variables. The variables created here, is known only in the statement. After the initialization the *for* statement test a condition, and if it is the *true*, executes the next block. If the condition is not *true*, the statement is ended, and the program continues with the next statement after the *for* statement. After the block have been executed the expression part is performed, and the condition is tested again. In the case where the block consists of only a single statement, it is as for the other control statements allowed to delete the parentheses { and }.

EXAMPLE: FACTORIAL

As an example is shown a command that has a single parameter. If it is a legal non-negative integer n, the command prints the factorial of n, where

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
…

That is, that *n!* is the product of all positive integers less than or equal to n.

```
package factorial;

public class Factorial
{
 public static void main(String[] args)
 {
  if (args.length == 1)
  {
   try
   {
    int n = Integer.parseInt(args[0]);
    if (n >= 0) System.out.println(fakultet(n));
    else System.out.println("The argument must be non-negative");
    System.exit(0);
   }
   catch (Exception ex)
   {
    System.out.println("The argument must be an integer");
   }
  }
  else System.out.println("Illegal number of arguments");
 }

 private static long fakultet(int n)
 {
  long t = 1;
  for ( ; n > 1; --n) t *= n;
  return t;
 }
}
```

Note first that the fatorial is calculated by the method *factorial()*, where the parameter is the number whose factorial to be determined. Otherwise consists the calculation of a *for* loop that calculates the result in the local variable *t*. You should note that the initialization is empty, what is allowed. In fact, all three parts of the *for* statement must be empty. In particular, if the condition part is empty, the condition is always *true*.

Most of the code in the *main()* method are used to validate the arguments from the command line. First *main()* tests whether there are the right number of arguments, and if it is not the case, the program stops with an error message. Is there an argument, the program will convert it to an *int*, and is the result a non-negative number, the method *factorial()* is called to calculate the factorial. In case of an illegal number the program prints an error message.

EXAMPLE: DIGETSSUM

I want to show another example of the use of a *for* statement which is also a command that is carried out with a single argument. Is this argument a legal non-negative integer, the command calculates the sum of the numbers digets and repeat that calculation until the sum is less than 10:

```
DigitsSum(123) = 1 + 2 + 3 = 6
DigitsSum(123456) = 1 + 2 + 3 + 4 + 5 + 6 = 21 = 2 + 1 = 3
DigitsSum(123456789) = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45 = 4 + 5 = 9
```

The *main()* method is in principle the same as above, so I'll just show the method, that calculates the sum of digets:

```
private static int digetssum(long t)
{
 for ( ; t > 9; )
 {
  long s = 0;
  for ( ; t > 0; s += t % 10, t /= 10);
  t = s;
```

```
 }
 return (int)t;
}
```

This time there are two *for* statements – one inside the other. In the outer *for* statement both the initialization and expression part are empty, and the loop is repeated so long that the variable *t* has more than one digit. The inner *for* statemment determines the sum *s* of the digits of the number *t*. Here you should note that

```
t % 10
```

means the last digit, while

```
t /= 10
```

throw the last digit away. Also note that the expression part this time consists of two expressions separated by commas, and the block is empty. As this the statement is probably not the most readable, but legal, it shows a little bit about how flexible the *for* statement is.

FOR AND ARRAYS

Given an array:

```
int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

it is probably the most classic use of a *for* loop to iterate through the array and do something with the individual elements. For example a loop which determines the sum of all elements in the array:

```
int sum = 0;
for (int i = 0; i < arr.length; ++i) sum += arr[i];
```

It corresponds exactly to what I have shown earlier when I introduced arrays. However, there is a variation of the *for* statement which is written as follows:

```
int sum = 0;
for (int t: arr) sum += t;
```

and is a loop doing exactly the same. This variant of the *for* statement works by menas of an iterator, but about this later. So far, the only advantage to write the statement on that way is a little shorter notaion, and perhaps the statement also is quite readable and easy to understand. In the literature this variant of the *for* statement sometimes is called a *foreach* statement.

EXERCISE 12

Write a program where you are in the *main()* method defines the following 2-dimensional array:

```
int[][] tal = {
 { 2, 3, 5, 7, 11},
 { 13, 17, 19, 23, 29 },
 { 31, 37, 41, 43, 47 },
 { 53, 59, 61, 67, 71 },
 { 73, 79, 83, 89, 97 } };
```

You must then add a method

```
private static int sum1(int[][] tal)
{
}
```

that as parameter has a 2-dimensional array. The method should determine and return the sum of the array's elements where the sum must be determined by means of two common *for* loops one inside the other.

Write a similar method

```
private static int sum2(int[][] tal)
{
}
```

that do the same, but this time the sum must be determined by two *foreach* statements.

EXERCISE 13

The Fibonacci numbers are a sequence of numbers where the first 11 is shown in the following table:

Index	Value
0	0
1	1
2	1
3	2

Index	Value
4	3
5	5
6	8
7	13
8	21
9	34
10	55

The first two (the 0'th and 1'th) Fibonacci numbers are defined as respectively 0 and 1. Then the numbers are defined in that way that a certain Fibonacci number is the sum of the previous two.

You must now write a program where the user should enter an integer. The program must then print all Fibonacci numbers whose index is less than or equal to the number entered.

PROBLEM 7

A prime number is an integer greater than 1, which is only divisible by 1 and itself. To determine whether a number is prime, you can try to divide by any number greater than 1 and less than the number to test for prime and examine whether there is a number that goes up. In this case, it is not a prime number. If none of the numbers go up, it is a prime number.

You can improve the algorithm by noting that 2 is the only even prime, and that it is enough to divide by number less than or equal to the square root of the number.

You should write a command that is executed with one argument on the command line. The argument must be an integer, and command should prints if the argument is a prime number.

5.4 THE *SWITCH* STATEMENT

while, *for* and *do* is control structures for loops, while *if* is a control structure for branching or selection. In most cases, *if* is the preferred statement for branching, but there is an alternative called *switch*. It is best explained with an example.

In the following program, the user must enter an integer. If the number is 1, 2, 3, 4, or 5, the program must print the name of the corresponding day, and if the number is 6 or 7, the program must print the text *Weekend*, and otherwise, the program must print an error message.

```
package weekday;

import java.util.*;

public class WeekDay
{
 public static void main(String[] args)
 {
  Scanner kb = new Scanner(System.in);
  System.out.print("Enter number of the day in the week: ");
  String text = kb.nextLine();
  switch (text)
  {
   case "1":
    System.out.println("Monday");
    break;
   case "2":
    System.out.println("Tuesday");
    break;
```

```
    case "3":
     System.out.println("Wednesday");
     break;
    case "4":
     System.out.println("Thursday");
     break;
    case "5":
     System.out.println("Friday");
     break;
    case "6":
    case "7":
     System.out.println("Weekend");
     break;
    default:
     System.out.println("Illegal day…");
   }
  }
}
```

When the program starts the user should enter a day's number. This is followed by a *switch* statement. After the word *switch* is an expression which here is merely a variable of type *String*:

```
switch(text)
```

The construction also has a number of *case* entries, where there after each *case* is a constant followed by colon. The constant must have the same type as the expression (which is here *String*):

```
case "1":
```

When the statement is executed, the expression is evaluated, and the control is transferred to the *case* entry which value is identical to the value of the expression. Hereafter are the next statements executed until one meets a *break* statement or the *switch* statement ends. If there is no *case* entry corresponding to the value of the expression, there are two options. If the *switch* statement has a *default* entry, control is transferred to it. Otherwise the entire *switch* statement is skiped. Note that a *switch* does not need to have a *default* entry. If in the above example 6 or 7 is entered, the program will in both cases print

```
Weekend
```

If, for example the user enter 6, the program will continue with the next statement after *case 6*, which are *case 7*, as there is no break after *case 6*.

The *switch* statement is kind of multi-branching, and it can sometimes be a sensible solution, but not as often as the other control statements. You should note that the type of the expression in *switch* can be all integer types, *char*, *boolean*, and *String*.

EXERCISE 14

You should write a program similar to the above, but you have this time to enter the number a month (1, 2, 3, …, 12). The program should print the season for the month:

- spring
- summer
- fall
- winter

5.5 *RETURN* STATEMENT

The *return* statement is already used many times, but it can also be perceived as a control statement. In general the statement interrupts a method and the control is passed to the place where the method was called. Note specifically that a *return* statement in the *main()* method interrupts the *main()* and thus stops the program. If the method is a *void* method there should not be anything after the *return*, and the effect is only that the method is terminated. If, however, the method has a type, then there after the *return* must be an expression of the same type as the method. The method will then return the value of this expression, a value which may then be sent back to the calling code. You should particularly be noted that a method may well have multiple return statements, typically controlled by an if statement.

5.6 *BREAK* AND *CONTINUE*

In conjunction with the *switch* statement, I have been using *break* as a statement that interrupts a *switch* statement. *break* may also be used to interrupt a loop.

Consider as an example the following method where *rand* is a random number generator, which is created at the start of the program (called *BreakContinue*):

```
private static void test1()
{
 int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
 int elem = rand.nextInt(29) + 1;
 int n = -1;
 for (int i = 0; i < arr.length; ++i)
  if (arr[i] == elem)
  {
   n = i;
   break;
  }
 if (n >= 0) System.out.println(elem + " found on place number " + n);
 else System.out.println(elem + " not found1");
}
```

The method creates an array of 10 integers, and a variable *elem* is initialized with a random number. The next *for* loop checks to see if this number is found in the array, and if so, save the index of the element in a variable *n*. When the item is found, there is no need to search far and I therefore wants to interrupt the loop. This can be done with *break*, and the method continues with the first statement after the loop.

The possibility to break out of a loop in this way is quite reasonable and has many uses. *break* can be used in the same way if the loop is *do* or *while*.

Consider then the following method, which just is a helper method in the program:

```
private static String create()
{
 String text = "abcdefghijklmnopqrstuvwxy1234567890";
 String line = "";
 for (int i = 0, n = rand.nextInt(30) + 10; i < n; ++i)
  line += text.charAt(rand.nextInt(text.length()));
 return line;
}
```

This method creates a random string whose characters consist of lowercase letters and digits. The string length is between 10 and 39 characters. You must specifically note the method *charAt()*, which is a method in the class *String*, which allows you to refer to the individual characters in a string.

```
text.charAt(rand.nextInt(text.length()))
```

is thus, an arbitrary character in the string

```
"abcdefghijklmnopqrstuvwxy1234567890"
```

There is a control statement that is called *continue* and is in many ways an allternative to *break*. Consider the following method:

```
private static void test3()
{
 String text = create();
 String tal = "";
 for (int i = 0; i < text.length(); ++i)
 {
  char c = text.charAt(i);
  if (c < '0' || c > '9') continue;
  tal += c;
 }
 System.out.println(tal);
}
```

The method creates a string of random characters – using the method *create()*. Then the method iterates through the string in a *for* loop, to build a substring consisting of all characters that are digits. Note again the use of the method *charAt()*. When in the loop there is a specific character, that it is not a digit:

```
if (c < '0' || c > '9') continue;
```

you can go to the next character, and here it is done with a *continue*. This means that controle is transfered to the end of the block, and hence skips the statement

```
tal += c;
```

continue is not used so often, but it has its uses. Note also that the problem in this case could be solved in other ways than using *continue*.

If one has a loop inside an other, and use *break*, it is only the inner loop, which is interrupted. If you wish to interrupt several levels, you can use a *label*. Consider the following method:

```
private static void test2()
{
 int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
 int n = -1;
 int elem;
 found:
 for (;;)
 {
  elem = rand.nextInt(29) + 1;
  for (int i = 0; i < arr.length; ++i)
   if (arr[i] == elem)
```

```
    {
     n = i;
     break found;
    }
  }
  System.out.println(elem + " found on place number " + n);
}
```

This method creates again an array with 10 elements. Next, it determines in the same manner as in *test1()* a random number, and searches for the number in the array. If the number is found, the index in the array is assigned to a variable *n*, but when the number is found, wee want to end the search, and that is done by a *break* to a *label*:

```
found:
```

It's just a name with a colon afterwards. The meaning is that control is transferred to the first statement after the *for* statement.

There are not so many "good" uses of *break* in this way, and you should generally consider whether it is a good solution as it can make the code difficult to read and understand.

Note the outer for loop:

```
for (;;)
```

It is an example of an infinite loop.

Also, the continue can apply a label. It is an opportunity that rarely used, but the following method shows how:

```
private static void test4()
{
 String text = "";
 for (int i = 0; i < 10; ++i) text += create() + "\n";
 next:
 for (int i = 0; i < text.length(); ++i)
 {
  String tal = "";
  for (; ; ++i)
  {
   char c = text.charAt(i);
   if (c == '\n')
   {
```

```
     System.out.println(tal);
     continue next;
    }
    if (c < '0' || c > '9') continue;
    tal += c;
   }
  }
}
```

The method starts by creating a string consisting of 10 substrings created with the method *create()* and separated by line breaks. Then the string is traversed from start to finish in a *for* loop, but this is preceded by a *label*. The loop has an inner loop that performs the same as in *test3()* and determines a partial string consisting of digits. You should note that the inner loop counts of (use) the same index as in the outer loop. If the inner loop encounters a line break, it prints the number that is determined, and then performs a *continue* to the label *next*. This means that the control continues at the end of the block of the outer *for* loop, and the next operation is performed after the outer loop counter is counted up by 1. The result of it all is that the method print any number (10 numbers) which appears as a substring in a line.

The code above is by no means beautiful and is also only included as an example of how to use *continue*. There is a reason to warn against excessive use of the *continue*, as the statement easily leads to code that is hard to read and understand.

PROBLEM 8

A palindrome is a phrase that is spelled the same front and rear, however, so that spaces and special characters are ignored, and that there is no distinction between uppercase and lowercase letters. Examples of (danish) palindromes are

- Otto
- En af dem der red med fane
- Madam, I'm Adam
- Selmas lakserøde garagedøre skal samles

You must write a command that is performed with one argument, and the command should test whether this argument is a palindrome.

PROBLEM 9

You must write a program that simulates a simple dice game called *Craps*:

A player rolls two cubes, and then the sum of the eyes are noted. If the sum is 7 or 11, the player has won. If the sum is 2, 3 or 12, the player has lost (the house has won). If the sum in contrast is 4, 5, 6, 8, 9 or 10, the sum is recorded as the player's points. The player then proceeds to throw the cubes until the sum of the eyes either are the player's points or until the sum is 7. Is the sum is the player's points, the player won. If the sum is 7, the player has lost, and the house wins.

The program must act in that way that the user first enter how many games he want to play. Then the program simulates the desired number of games, and last prints, how many times the player has won, and how many times the house has won. It must be repeated until the user enter 0 for number of games.

There is however some design requirements for the program since it must consist of several classes. The task must be solved according to the following guidelines:

1. Start by creating a NetBeans project that you can call *Craps*.
2. Add a class *Cube* to the project – the class must be completely identical to the corresponding class from chapter 3, so you can copy the code from there.
3. Add a class *Cup* to represent a cube cup. The class must be equal the corresponding class from the project Cubes, so you can start by copying the code from this class. The class must then be changed as follows:

   The method *yatzy()* should be removed.

   Add a method

   public int count()

   that returns the number of cubes in the cup.

   Add a method

   public Cube getCube(int n)

   that returns the n'th cube in the cup.

4. Add a class *Game*, which represents the game itself, that is simulates that the game is played a certain number of times and prints the result.
5. Write the code for the *main* class where the user can enter the number of times the game should be played.

When finished, it is important that you test the program well. You can not do much else than excutes the program many times and evaluate the result.

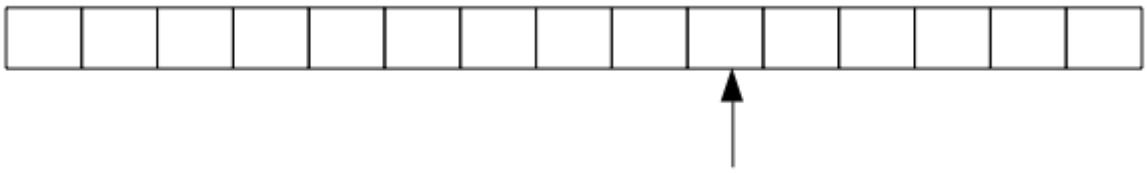
The above version of the game is somewhat simplified, and there are more rules associated with the game. If you play the game, as described above, the player and the house has almost the same probability of winning with a small overweight to the house – and that it should also like to be if the house not has to go bankrupt.

6 ARRAYLIST

Java has a family of so-called *collection classes*, which are classes that can be thought of as containers for objects. These classes are discussed later (in the book Java 4), but I will introduce one of them, as you very often need. The class is called *ArrayList*. Yes indeed it is difficult in practice to write programs without having this class available.

A key concept in Java and, for that matter in any other programming language is an array, which is a data structure that may contain a number of objects of a certain type. The use of arrays is in principle quite straightforward, and there is rarely associated with the major problems to it, but there's one problem, namely that at the time when you create an array, you have to say how many objects there must be room for. It is far from always possible, and in many cases you have to solve the problem by creating arrays that are large enough. In other contexts, it may be necessary to count how many elements an array must have room for, and so then create an array of the right size. It can lead to inefficient code, and it is precisely these difficulties that the class *ArrayList* can solve.

Technical is an *ArrayList* a class which encapsulates an ordinary array, and the class has methods for manipulating the data in the array. One should think of an *ArrayList* as illustrated below, where the arrow indicates the next available place:



The basic method is called *add()*, and it adds an object to where the arrow points and after the object is added the arrow is moved one place right. When you create a new *ArrayList* the arrow is at the location to the extreme left, corresponding to that the list is empty. As added objects, move the arrow to the right, and attempts to add an object, and there is no space, the array automatically expands. That is exactly what is the data structure's strength, and as a programmer you can fill objects into the list without considering whether there is room or not. The class has many other methods, so as an example you can refer to the individual objects in the list with a method *get()* and you can by an index insert elements in the middle of the list, delete items, etc., but it is important to bear in mind that an *ArrayList* is a sequence, and that it can not have holes. This means in practice that if you insert an element in the middle of the list, all elements to the right of the new element is moved one place, and in the same way if you delete an element that then all elements to the right of the item that is deleted has to move one place left. In general are the class and its methods effective, and specifically is *add()* extremely effective, and as a programmer you should not think about all the technical stuff, but think of the class as a dynamic array. As an example the following program creates an ArrayList:

```
public static void main(String[] args)
{
 ArrayList<String> list = new ArrayList();
 list.add("Gorm den Gamle");
 list.add("Harrald Blåtand");
 list.add("Svend Tveskæg");
 list.add("Knud den Store");
 list.add(3, "Harrald d. 2.");
 System.out.println(list.get(2));
 for (String s : list) System.out.println(s);
}
```

You should note the syntax of how to create the list, and how to specify that the list should be used for *String* objects. You should specify what it is for a kind of elements that the list should contain. Next the program add 5 elements to the list, but the last *add()* has an index that indicates where you want to insert the element. The second last statement shows how to refer to an element with *get()*. Finally, the last statement, shows how to iterate through the items in the list. You must also be aware of the method *size()*, which is not used in this example, but returns the number of elements in the list, and you could for example iterate through the list as follows:

```
for (int i = 0; i < list.size(); ++i) System.out.println(list.get(i));
```

The class *ArrayList* is easy to use, and I will in the following apply it where it is needed and especially I will apply it in the final example.

7 COMPARISON AND SORTING

If you executes the following method

```
private static void test01()
{
 String s1 = new String("Knud");
 String s2 = new String("Knud");
 System.out.println(s1);
 System.out.println(s2);
 System.out.println(s1 == s2);
 System.out.println(s1.equals(s2));
}
```

you get the result:

```
Knud
Knud
false
true
```

and here is the result *false* hardly what you would expect. The explanation is that *s1* and *s2* are references to two different objects, and comparing two objects with the equal sign it is the object references that are compared. If instead you want to compare the values of the two objects, that the variables referennces you must use *equals()* as in the last sentence.

Consider as an example the following class that represents the name of a person:

```
package comparison;

public class Name
{
 private String firstname;
 private String lastname;

 public Name(String firstname, String lastname)
 {
  this.firstname = firstname;
  this.lastname = lastname;
 }

 public String getFirstname()
 {
  return firstname;
 }
```

```
 public String getLastname()
 {
  return lastname;
 }

 public String toString()
 {
  return firstname + " " + lastname;
 }
}
```

If you executes the following method

```
private static void test02()
{
 Name n1 = new Name("Olga", "Jensen");
 Name n2 = new Name("Olga", "Jensen");
 System.out.println(n1);
 System.out.println(n2);
 System.out.println(n1.equals(n2));
}
```

you get the result:

```
Olga Jensen
Olga Jensen
false
```

where the comparison of the objects – though it is done with *equals()* – gives *false*. It is not entirely what has been said above, but the explanation is that *equals()* as default works in the same way as comparing with equality, and should *equals()* compare the objects' values, you must implement the method:

```
public boolean equals(Object obj)
{
 if (obj == null) return false;
 if (obj.getClass() == getClass())
 {
  Name n = (Name)obj;
  return firstname.equals(n.firstname) && lastname.equals(n.lastname);
 }
 return false;
}
```

After this the comparison will give the right result. That the method *equals()* should be written as shown above, you should just accept it in this place, but the following happens:

- if the parameter is *null* the two objects can not be *equals()*
- if the two objects are instances of the same class, the parameter is converted to a *Name*, and the two objects are equal if they has the same first name and the same last name
- else the two objets can not be *equals()*

When the comparison of *firstname* and *lastname* (and incidentally also the method *test01()* works), it is because the *String* class implements *equals()* so that it compares the values.

Consider the following method:

```
private static void test03()
{
 String[] navne = { "Svend", "Knud", "Valdemar", "Harald" };
 for (String s : navne) System.out.print(s + ", ");
 System.out.println();
 Arrays.sort(navne);
```

```
 for (String s : navne) System.out.print(s + ", ");
 System.out.println();
}
```

If you executes the, you get the following result:

```
Svend, Knud, Valdemar, Harald,
Harald, Knud, Svend, Valdemar,
```

This example shows that it is quite simple to sort the array, since the class *Arrays* has a *sort()* method. Executing the following method:

```
private static void test04()
{
 Name[] names = { new Name("Svend", "Grathe"), new Name("Knud", "D. 5."),
   new Name("Valdemar", "Den store"), new Name("Harald", "Blåtand") };
 for (Name n : names) System.out.print(n + ", ");
 System.out.println();
 Arrays.sort(names);
 for (Name n : names) System.out.print(n + ", ");
 System.out.println();
}
```

however, it will fail, and the program stops with an exception. The reason is that objects not in general can be compared and be arranged in a sequence. It supports the class *String*, but for your own classes you must define the comparison – and it is also natural, for the system obviously can not know how custom objects should be arranged. If it should be possible to compare objects of the type *Name*, the class must implements an interface:

```
package comparison;

public class Name implements Comparable<Name>
{
 private String firstname;
 private String lastname;

  public Name(String firstname, String lastname)
 {
  this.firstname = firstname;
  this.lastname = lastname;
 }
```

```
 public String getFirstname()
{
 return firstname;
}

 public String getLastname()
{
 return lastname;
}

public String toString()
{
 return firstname + " " + lastname;
}

public boolean equals(Object obj)
{
 if (obj == null) return false;
 if (obj.getClass() == getClass())
 {
  Name n = (Name)obj;
  return firstname.equals(n.firstname) && lastname.equals(n.lastname);
 }
```

```
  return false;
 }

  public int compareTo(Name n)
 {
  if (lastname.equals(n.lastname)) return firstname.compareTo(n.lastname);
  return lastname.compareTo(n.lastname);
 }
}
```

The class must implement an interface called *Comparable<Name>*, which says that it is possible to compare objects of the type *Name*. The interface defines a single method called *comparaTo()*, which is implemented at the end of the class. The method has a parameter of the type of the objects to be compared, and the method is implemented so as to meet the following protocol:

1. if the current object is less than the parameter, the method must return a value less than 0
2. if the current object is greater than the parameter, the method must return a value greater than 0
3. if the current object is equal to the parameter, the method must return 0

The class *String* implements this interface, and thus the method *test03()* works. In this case I would compare the *Name* objects such that if the two objects have the same last name, it is the first name that determines the order, otherwise the last name.

After the class *Name* is changed to implements *Comparable<Name>* also the method *test04()* works:

```
Svend Grathe, Knud D. 5., Valdemar Den store, Harald Blåtand,
Harald Blåtand, Knud D. 5., Valdemar Den store, Svend Grathe,
```

The above examples shows that it is simple to sort an array. The only requirement is that the objects to be sorted, implements the interface *Comparable*. Also objects in a *ArrayList* can be sorted, and the following method will sort 4 *Name* objects:

```
private static void test05()
{
 ArrayList<Name> names = new ArrayList();
 names.add(new Name("Svend", "Grathe"));
 names.add(new Name("Knud", "D. 5."));
 names.add(new Name("Valdemar", "Den store"));
 names.add(new Name("Harald", "Blåtand"));
 for (Name n : names) System.out.print(n + ", ");
 System.out.println();
 Collections.sort(names);
 for (Name n : names) System.out.print(n + ", ");
 System.out.println();
}
```

You should note that the only difference is that the sorting method *sort()* this time is a method in the class *Collections*.

8 FILES

Files are first explained in the book Java 5, but because I use files in the following chapter and also in the following books I will at this location show a little bit about what it takes to use a file – and it is not necessarily much.

From the perspective of the system a file is a sequence of bytes, and one can in many ways think of a file as an infinitely large array, where the first byte has index 0:

Of course, a file will not be infinite and there is an upper limit determined by the operating system and disk space, but in practice it is limitations that you almost always will ignore. When data is saved in a file and, above all, when data is to be read again, sequences of bytes must be converted from and to the data objects used by the application, and you must specify where in the file to write or read. It's all that kind of things that the Java IO classes take care of, and for many file operations goes all by itself – when you just know what you has to write.

In the following I will only show how one can write and read text files, and how to write any object to a file and read it again. The last is also known as object serialization / deserialization. In both cases it is simply a matter of accepting the syntax of what to write, but if you can manage text files and serialize objects to a file, you also know most of what is necessary to use files in your programs.

8.1 TEXT FILES

A text file is a file that contains usual lines of text separated by carriage returns. Seen from Java you use such files by writing and reading one line at a time. If you have to write to a text file, the process is:

1. open the file
2. create the line to be written to then file
3. write the line to the file
4. repeat this two operations as long as there are more lines
5. close the file

An example is shown a method *createLine()* that creates a line. The line consists of random integers (between 0 and 99) separated by a semicolon.

```
private static String createLine()
{
 String line = "" + rand.nextInt(100);
 for (int i = 0, n = rand.nextInt(10); i < n; ++i)
  line += ";" + rand.nextInt(100);
 return line;
}
```

The number of integers at each line is random, and a line will contain at least one number (and max 10), but otherwise there is nothing to explain.

The next method writes 10 such lines in a file:

```
private static void test1()
{
 try
 {
  BufferedWriter writer = new BufferedWriter(new FileWriter("numbers"));
  for (int i = 0; i < 10; ++i)
  {
   writer.write(createLine());
   writer.newLine();
  }
  writer.close();
 }
 catch (Exception ex)
 {
  System.out.println(ex);
 }
}
```

The method works as described in the above algorithm, and there is not much other than to take note that this is how to write, but you must be aware of the following:

- *BufferedWriter* is a class that represents a text file width the name *numbers*. The first statement opens the file for writing, and in the program the file is called *writer* (the variable that references the file).
- You write a line in the file with the method *write*, and in this case the line written to the file is the line created by *createLine()*.

- The method *newLine()* writes a line feed in the file.
- The method *close()* close the file, and it is important as it may first be at that time the data is physically written to the file.
- The whole is placed in a try/catch, and it is necessary, when file operations (methods for file processing) typically can raise an exception.

You should note that in principle it is the same thing that will happen every time you have to write lines to a text file. It is only the method *createLine()* that will vary.

On the disk the file is called *numbers*, and it is saved in the application's current directory. You should also note that if you executes the program several times, it will every time overwrite an existing file, if there is a file with the same name. If you executes the method, the result could be a file containing the following:

```
49;42;38;69
34
43;85;60;77;91
49;60;16;36;9;3;41;62
72;90;34;50;69;37;18;27;80;90
71;69;42;51;21
47;42;67;35;19;52;29;54;17;74
```

```
62;77
87;94;39;94
69;32;2;69;79
```

After creating the file is the next step to show how to read the content of the file. In principle, the algorithm is:

1. open the file
2. read the next line from the file
3. process the line in the program
4. repeat this two operation as long there are more lines in the file
5. close the file

It is thus almost the same algorithm as when the file is written. In this case, I read all the lines in the file and calculatte the sum of the numbers contained in the file.

When you read a line the result is text and the line should therefore be split and each element must be converted to a number. In addition, I have written the following method – with reference to the algorithm is similar to *process the line in the program*:

```
private static int parseLine(String line)
{
 int sum = 0;
 String[] elems = line.split(";");
 for (String elem : elems) sum += Integer.parseInt(elem);
 return sum;
}
```

The method is simple enough and contains nothing new, but note, however, the method *split()*, which is a method defined in the class *String*, that splits a string into substrings when the substrings are separated by semicolons. The method returns the sum of the numbers in a single line.

The following method reads the content of the file and determines the sum of the numbers:

```
private static void test2()
{
 try
 {
  int sum = 0;
  BufferedReader reader = new BufferedReader(new FileReader("numbers"));
  for (String line = reader.readLine(); line != null; line = reader.readLine())
   sum += parseLine(line);
```

```
  reader.close();
  System.out.println(sum);
 }
 catch (Exception ex)
 {
  System.out.println(ex);
 }
}
```

Also, this method follows the algorithm. This time the file is represented by an object of the type *BufferedReader*, which is an object that opens a file for reading. The subsequent *for* loop starts with in the initialization reading the first line of the file. If it goes well – the variable *line* is different from *null* – the line is processed, which is done by calling the method *parseLine()*. Next, read the next line (the expression part), and it repeats itself until the method *readLine()* returns *null*. It happens when you have reached the end of the file. Once all lines are read, the file is closed, but in conjunction with reading files it is not quite as important as with writing files, but a good principle.

If the method is performed, the result could be:

```
2825
```

The above example, called *TextFiles*, shows what it takes to write and read a text file, and even though there is a lot more, the above is sufficient to use simple text files in practice.

EXCERCISE 15

You should write a program similar to the above. The program must similarly have two test methods.

The first method must open a file for writing. Next it must perform a loop where the user in each iteration must enter a name and the name should be writen to the file. The iteration stops when the user just types enter to the name.

The second method should read the content of the file and should only prints the names on the screen.

8.2 SERIALIZATION OF OBJECTS

It is also possible to store objects in a file, and actually it is not quite simple. When a file is of course just a sequence of bytes, this means that the object and its data have to be streamed to a sequence of bytes, a process that is called for *serialization*. It might not be so difficult, but the object must be able to be read again, and it must be possible to convert a sequence of bytes into an object of one type or another. It is only possible if together with the object is saved information about its type and serialization must not only stream object's data, but also information on the type. With this information it is possible to restore an object from a sequence of bytes, a process that we call *deserialization*. It sounds complicated, and it is that too, but seen from the programmer, it is simple, because Java has the necessary classes, which takes care of it all.

I will write a program that stores an *ArrayList* with objects of the type *Name*, where it is the same class that you have seen in chapter 7, however, with one change:

```
import java.io.*;

public class Name implements Comparable<Name>, Serializable
{
```

For an object to be serialized, its class must implement the interface *Serializable*. It is an empty interface, which does not define any methods or contains anything else, and such an interface are sometimes referred to as a *marker interface*. Indeed, it may sound strange, what it should do well, and you must in this book just take note of the need to implements that interface if the objects must be able to be serialized.

Consider the following method, which creates an *ArrayList* with 4 *Name* objects:

```
private static ArrayList<Name> createNames()
{
 ArrayList<Name> list = new ArrayList();
 list.add(new Name("Karlo", "Jensen"));
 list.add(new Name("Gudrun", "Andersen"));
 list.add(new Name("Valdemar", "Hansen"));
 list.add(new Name("Olga", "Knudsen"));
 return list;
}
```

The method requires no explanation, but the goal is to serialize the *ArrayList* that the method returns. It is possible, because also the class *ArrayList* is defined *Serializable*, and because the list contains *Serializable* objects, the entire structure can be serialized as an object. This is done as follows:

```
private static void test1()
{
 try
 {
  ObjectOutputStream stream =
   new ObjectOutputStream(new FileOutputStream("names"));
  stream.writeObject(createNames());
  stream.close();
 }
 catch (Exception ex)
 {
  System.out.println(ex);
 }
}
```

ObjectOutputStream represents a file, to which you can write objects, and an object is written using the method *writeObject()*. Simpler it can hardly be. In this case, the returned object from *createNames()* is written, and the object is stored in a file called *names*. You should note that the content of the file this time is not clear text, and an attempts to open it, you get a result that you can not easily understand.

As a next step, I will show how to deserialize the object. Consider first the following method that prints the contents of an *ArrayList* on the screen:

```
private static void printNames(ArrayList<Name> list)
{
 for (Name n : list) System.out.println(n);
}
```

The object can then be deserialized as follows:

```
private static void test2()
{
 try
 {
  ObjectInputStream stream = new ObjectInputStream(new
FileInputStream("names"));
  printNames((ArrayList<Name>)stream.readObject());
  stream.close();
 }
 catch (Exception ex)
 {
  System.out.println(ex);
 }
}
```

Again, there is not much to explain, but you can notice that the file from where to load objects are represented of an object of the type *ObjectInputStream*. The object is read with the method *readObject()*. It returns an *Object*, and it is therefore necessary with a typecast to convert the object to an *ArrayList<Name>*, before it is sent to the method *printNames()*. You must specifically note the syntax for a typecast, where the type to convert to, is put in parenthesis:

```
printNames((ArrayList<Name>)stream.readObject());
```

EXERCISE 16

You must write a program similar to the above with a test method to serialize an object and another test method to deserialize the object. Start by writing a method

```
private static int[] createNumbers()
{
}
```

that creates and returns an array of the type *int[]* with 100 numbers when it must be initialized with the values 1, 2, 3, …, 100.

Then write a method *test1()* that serialize such an object to a file, that is an object returned from *createNumbers()*.

Write a method:

```
private static void printSum(int[] arr)
{
}
```

that prints the sum of the numbers in an array on the screen. Write then a method *test2()* to deserialize the object from before (the array with 100 numbers) and prints the sum of the numbers on the screen. The result must be 5050.

9 FINAL EXAMPLE

At the end of the first book I will show a slightly larger example, where I as mentioned in the forword mainly will focus on how the program should be written and less on the code. For the code I refer to the finished program, which incidentally is fully documented.

The task is to write a program that can print lottery tickets on the screen, but a program that also can be used to from the week's lotto numbers to determine the number of correct rows. The program should be used as a command from a terminal where the user with options specify what the program should do. A typical use of the program is that the program is used to create the rows you want to play. When the week's winning numbers are drawn, you then use the program to determine the game's outcome. There are several (many) kinds of lottery, examples include a lottery with 36 numbers and a lottery with 48 numbers that are examples of Danish lottery games. The program must be written so that it can be used for all lottery games which is of that nature.

A lottery game consists of a number of lottery rows, where a row consists of a number of different lottery numbers that are integers within a range, and in this program, a lottery game is characterized by the following parameters:

-a	the minimum allowable number	1
-b	the maximum allowable number	36
-r	number of lottery numbers at a row	7
-n	number of rows to play	

where the last column is a default value that corresponds to the ordinary Danish lottery. These values can be obtained as options on the command line. You can also use the following options:

-o	name of the output file for the result, where the default is the screen
-i	name of the input file when the program is used to check the week's lottery
-u	the week's lottery numbers separated by spaces

The last two options are used to check the lottery numbers of the week, while the other five options are used to create lottery rows, that is a new game. The program is called *Lotto*, and below are examples of legal commands:

```
java -jar Lotto.jar -n 20
java -jar Lotto.jar -a 1 -b 48 -r 6 -n 20 -o uge42
java -jar Lotto.jar -i uge42 -u 2 13 18 22 34 35
java -jar Lotto.jar -i uge42 -u 2 13 18 22 34 35 -o uge42-res
```

Generally, there are the following requirements for the arguments on the command line:

- All options which have a default value (*a*, *b*, *r*, *o*) may be omitted.
- Options may appear in any order.
- There must be no conflicting options (for example *-r* and *-i*) on the same command line.

The concept of a lottery ticket is not included in the solution as it in this program does not make sense – a lottery game is basically one large coupon that can have any number of rows.

In terms of the result of a lottery there are the following requirements:

1. The numbers in each lotto row should be sorted in ascending order.
2. The same row must not occur several times.
3. The rows should be sorted in ascending order (after the first number, after the next number, etc.).
4. The first line in the result defines the game and contains the smallest and largest legal lottery number, and the number of values in a row and the week number.

With regard to the results of the verification it is a file (the screen by default) that contains all rows, but where each row is added a number indicating the number of correct values. The result is sorted by this number in descending order.

If you enter an illegal command, for example because it is wrong, missing or conflicting options, illegal lottery numbers, illegal files, etc., the program must terminate with an error message and the program syntax. If checking the week's lottery there is an illegal row, the number after the row should be replaced with a short error message, and possibly illegal rows should be added last in the result.

Boundaries for the program's parameters are:

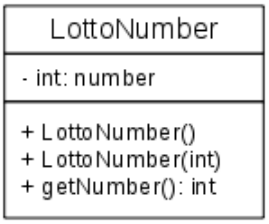
$$a \geq 1 \wedge a < 100$$
$$b \geq a + 5 \wedge b < 100$$
$$r > 2 \wedge r \leq 15$$
$$r < b - a$$

Boundaries for the program's parameters areThat is a lottery number maximum can have two digits, and a lottery row can have up to 15 numbers. In the same way there are some minimum requirements. There are really not many reasons for these requirements in addition to making it easier to formats the result and avoiding some special cases, such as the game can be empty.

9.1 DESIGN

The above can be seen as a specification of requirements for the program to be written. The program can be written in many ways, and from the one that use the program it plays no great role, as long as the program meets the requirements, but in practice, programs must be maintained, and here it is very important how the program is written. It is important that the program is written in such a way that the code is easy to read and understand, and it is important that the code is written so that a change in one place does not mean that large parts of the program must be rewritten. To meet these goals the program must be organized into good classes, and the question is what it is and how to find and write these classes. It is not simple, and there is not a unique solution, and this is largely what the following books is about, but below will explain a little about how this program is organized into classes, and thus the program's architecture.

If one considers the concept of a lottery row, then it is a sequence of different integers (lottery numbers) within a range. The lotto number is nothing more than an *int*, but I have nevertheless chosen to define a class to represent a lottery number. The reason is that the class must be able to validate a lottery number and check that it is legal in relation to the current lotto game, and the program should therefore have the following class:



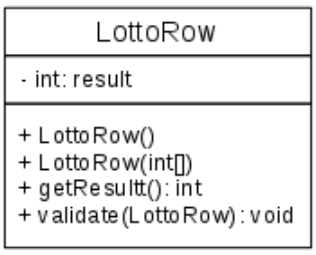
Often you will in system development illustrate the classes that way. It says that the program must have a class called *LottoNumber*, and this class must have a private *int* variable called *number*. In this case, this number represents the value of a lottery number. The figure also shows that the class should have two constructors, where the first is a default constructor that should create a *LottoNumber* object with a random legal value, while the other has the value as a parameter. It must be used when the program creates the week's lottery. In addition the figure shows that the class must have a method *getNumber()* to returns the lottery number's value.

The class *LottoNumber*, must as mentioned validate a lottery number and should therefore know what are legal values. In this case, it is solved by defining the following class

The class should have a method *init()* with three *int* parameters which indicates the minimum legal lottery number, the largest legal lottery number and the number of lottery numbers in a row. The class has methods that can return these values. In the figure, the methods are shown underlined, which means that they are *static* methods and can be used without having an object. The details concerning static and non-static members in a class is explained in the book Java 3, and here it is enough to know that static methods in a class can be used anywhere in the application without having any object. Static methods are referenced using only the class name. The solution was chosen because the three values concerning a concrete lotto game must be available throughout the program. It should be emphasized that it is not the best solution, but it is the solution that is feasible with what until this place is said about Java.

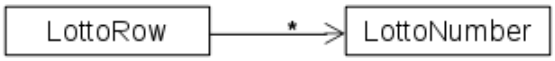
With this class available, it is clear that the class *LottoNumber* and its methods can validate whether a lottery number is correct.

The two classes *LottoNumber* and *LottoGame* is both simple, but the next class is more complex:



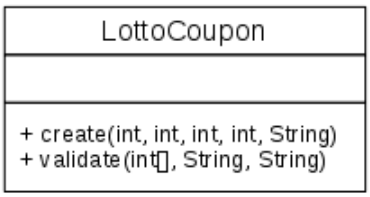
The class represents a lottery row, and thus a sequence of lottery numbers. The class has two constructors, where the first must create a lottery row with random lottery numbers, while the other creates a lottery row with specific numbers and should be used to represent the week's lottery. Both constructors must ensure that the lottery row created is legal in relation to the current lottery game, and therefore have to use the class *LottoGame*. In particular, is the first constructor complex since it must ensure the creation of a lottery row where all the numbers are different, and that the row's numbers are sorted in ascending order. The class should have a method *validate()* with a parameter that is a *LottoRow*. This parameter represents the week's lottery, and the method should be used to determine how many correct numbers there are on the current row. The result is that the method updates the *result* variable, and when it is private, there is also a method that can return the value.

The class *LottoRow* must consist of a number of *LottoNumber* objects, but instead of showing it as a variable in the class's figure, we show it as a link between the two classes:



Here the * tells that a *LottoRow* object refers to several *LottoNumber* objects.

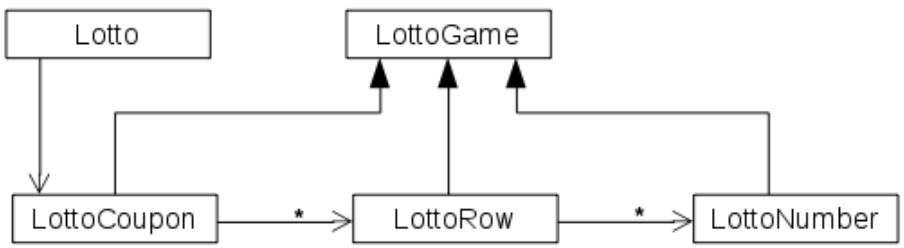
There is also a need for a class that may represent a number of lottery rows and thus a particular lottery game. Although I opposite has mentioned that the program not uses a concept similar to a lottery coupon, I have decided to call the next class for *LottoCoupon*, and one can think of the class as a concept similar to a lottery ticket of any size:



The class has only two methods, but they are, in turn, complex both. The first must create a new lottery game, and it has this five parameters, where the first three are the game parameters (minimum lottery number, biggest lottery number and the number of lottery numbers in a row), next is the number of rows to be played, and finally the name of the file that will contain the result. With the class *LottoRow* available is in principle simple enough to create the wished number lotto rows, but the method must ensure not to create two identical rows, and the rows must be sorted before being written to the file. Finally, the method *create()* will write the rows to the file, and here it was decided that the first line should be the game's parameters and week number. The method *validate()* is used to validate a lotto game against the weekly lottery. The method has three parameters, the first are the numbers for the week's lottery, while the next is the name of the file with the games to be validated, and the last is the name of the file to the result. The method then should both read data from a file and write data to a file. The most difficult is the of course validating of the individual rows. The class *LottoCoupon* is definitely the program's most complex class.

For the last is the class *Lotto*, which is the class with the *main()* method. In principle, it is a simple method because it simply must create a *LottoCoupon* object and call one of the methods. Yet fills the code much, because the *main()* method must parse the command line, and this time there are many possibilities for the arguments. Finally, it is also the *main()* method that has to print an error message in case of an error.

The program consists of a total of 5 classes, and the relationship can be illustrated by the following diagram:



9.2 PROGRAMMING AND TEST

I will not show the code for each class here, but refers instead to the final solution. You are encouraged to study the code in detail and focus on how the methods are written and works, and there are many lines of code to address.

When a program is finished, it must be tested. I would in other books specifically look at the test, which is not all so easy again, but in this case you must test the software with different combinations of options and check whether you get the expected result, and it is important to test all combinations and also combinations that result in an error.

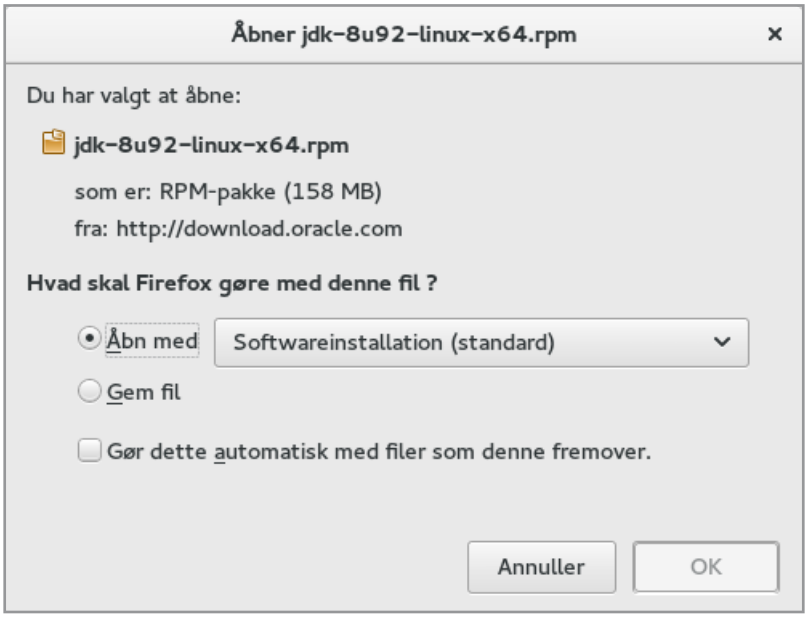
The program *Lotto* is as mentioned considerably larger than the examples which I otherwise dealt with in this book, but there is still talk about an example for the program's own fault, and it's not a program that has no practical justification. There is still more about Java you have to learn to be able to write programs with value in the real world, and especially lacks what it takes to write a program with a graphical user interface. It is the subject of the next book in this series. When you study the program *Lotto*, you must especially think of the program as an example that consisting of several classes and how those classes jointly solve the current problem. Moreover, you can consider the example as a prototype of what is meant by a command, and how to write a command.

APPENDIX A

The following shows how you can install Java and NetBeans, if it is not already on the machine. Start by downloading Java jdk (JDK for *Java Development Kit*). It can be downloaded from the following address:

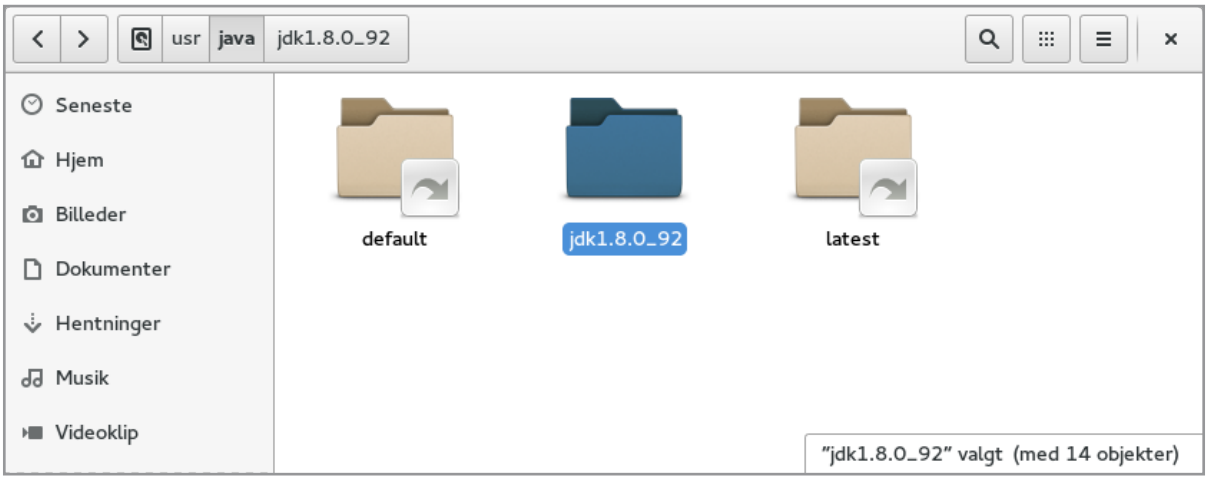
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Select the latest version and select the one that suits your operating system. I run a 64-bit Fedora, and I have therefore chosen *jdk-8u92-linux-x64.rpm*:

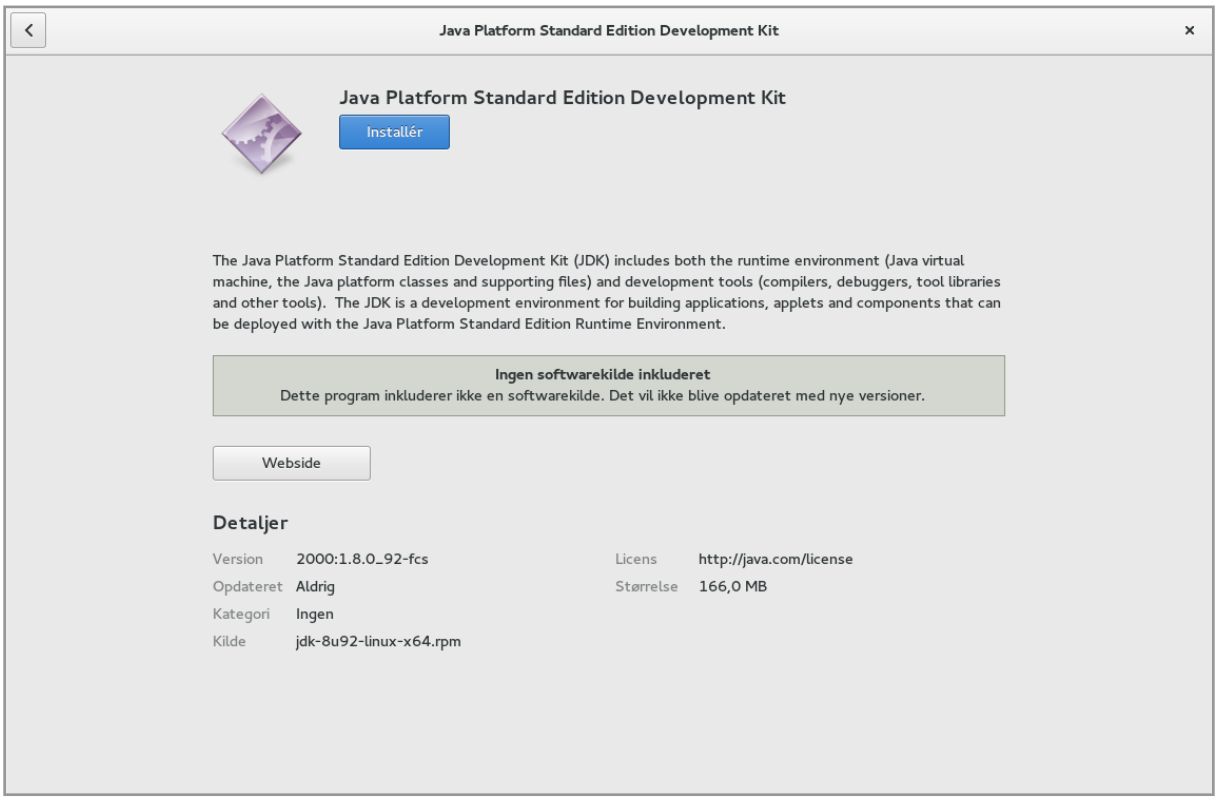


I have chosen to open the file with standard *Softwareinstallation*, and after download I get the window shown belove.

Then I click on the Install button, and Linux automatically installs *Java* and *jdk* the right place:

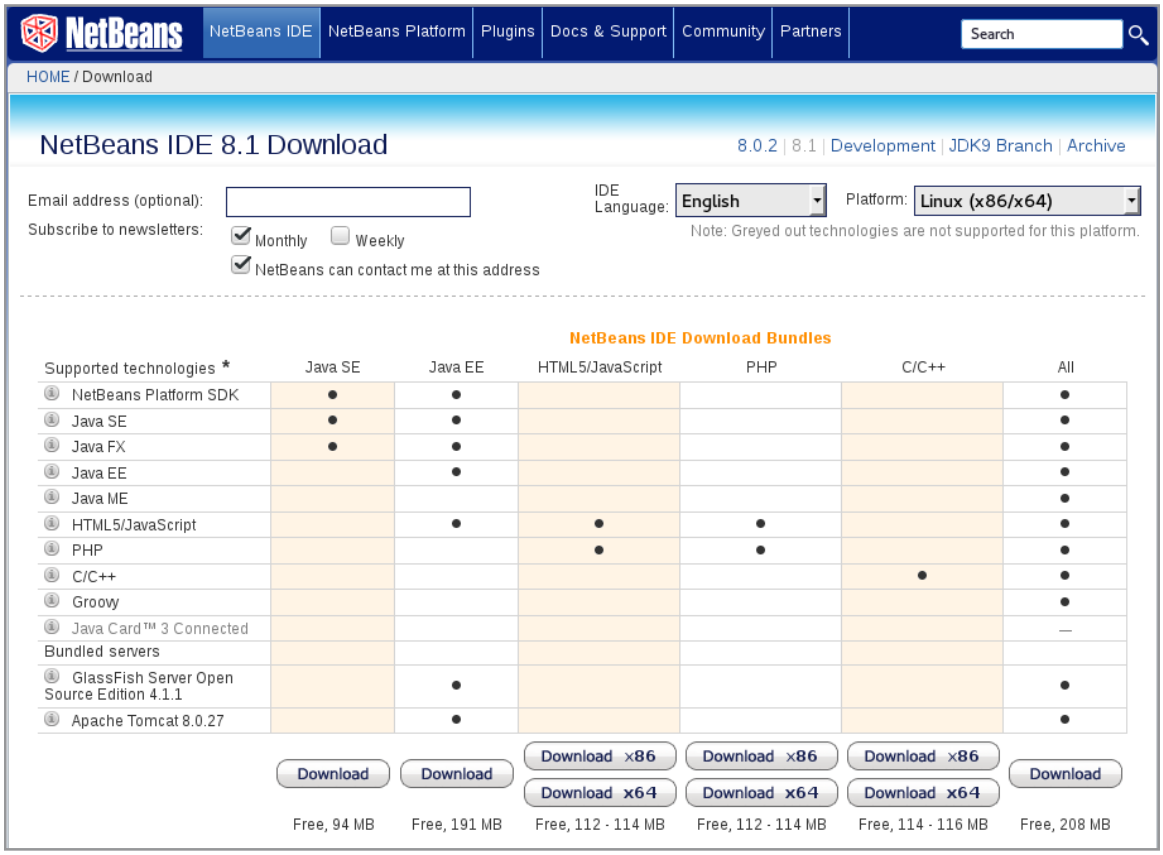


After that Java is installed.



As the next step you should install the development tool NetBeans. It can be downloaded from

https://netbeans.org/downloads/


NetBeans
NetBeans IDE
NetBeans Platform
Plugins
Docs & Support
Community
Partners
Search
HOME / Download
NetBeans IDE 8.1 Download
8.0.2 | 8.1 | Development | JDK9 Branch | Archive
Email address (optional):
Subscribe to newsletters:
Monthly
Weekly
NetBeans can contact me at this address
IDE Language:
English
Platform:
Linux (x86/x64)
Note: Greyed out technologies are not supported for this platform.
NetBeans IDE Download Bundles
Supported technologies *
Java SE
Java EE
HTML5/JavaScript
PHP
C/C++
All
NetBeans Platform SDK
Java SE
Java FX
Java EE
Java ME
HTML5/JavaScript
PHP
C/C++
Groovy
Java Card™ 3 Connected
Bundled servers
GlassFish Server Open Source Edition 4.1.1
Apache Tomcat 8.0.27
Download
Download x86
Download x64
Free, 94 MB
Free, 191 MB
Free, 112 - 114 MB
Free, 112 - 114 MB
Free, 114 - 116 MB
Free, 208 MB

Here there are several options, but for the sake of this and the next several books the first Java SE is sufficient, and I would recommend that you download it. Later in the context of web applications, you must use Java EE, but I would recommend you first update your IDE at that time since it requires installation of an *Application Server*, and there is no reason to have it running before you have the need.

When the product is downloaded, you've got an installation script that you must perform. However, you must first change the rights, so the script can be executed. Find the script with *Files*. Right-click the icon and choose *Properties*. Here, select *Privileges* tab and tick off *Execution*.

Then open a Terminal, and set current directory to the folder that contains the script. The product can now be installed with the following command:

```
sudo sh netbeans-8.1-javase-linux.sh --javahome /usr/java/jdk1.8.0_92/
```

or what your files is called. There may be differences because of the version numbers. The command installs the program and will open more windows including to indicate where the product is to be installed. For each window you simply accept the default, then the application is installed on the machine and a desktop icon is created.

After Java and NetBeans are installed, you can perform all examples and exercises described in this book.

However, there is one challenge back, otherwise you will encounter problems in the next book in the development of GUI programs. From a Terminal execute the following command:

```
sudo dnf install java-1.8.0-openjdk
```

It may take some time, but then everything is installed.