Poul Klausen

JAVA 5

Files and Java IO

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 5: FILES AND JAVA IO

SOFTWARE DEVELOPMENT

Java 5: Files and Java IO: Software Development
1st edition
© 2017 Poul Klausen & bookboon.com
ISBN 978-87-403-1735-0
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the fifth in a series of books on software development. The programming language is Java, and the language and its syntax and semantic fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. As the previous book, this book has, however, only to a lesser extent focus on the process, but more on the language and details regarding Java. The subject of this book are files and what Java provides to IO, and the goal is in details to show how to manipulate files. Although the files today do not means exactly the same for practical programming, the processing of files is yet knowledge which is required to have in order to be able to work as a professional software developer. Finally, it is important to be aware that IO plays a crucial role in computer networking, and much of what follows will be used and expanded in the book Java 15 on network programming. Furthermore treat this book data representation, which are mostly deferred to an appendix, which also gives a brief introduction to the binary and hexadecimal numbers. The detailed data representation do not play a big role in everyday life, but also here is a topic that is necessary to have knowledge of, for example, in the context of computer networks. The book assumes that the reader has a knowledge corresponding to what is addressed in Java 3 and Java 4, but if you wish, the book can be postponed until you get the need.

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

Many applications need to store data persistently and later read them again. As an example can you think of a word processor, where the user is editing a document. Here, the program could save the document for later load it again, so you can edit it further. As another example, one can think of software in a terminal at the supermarket that must be able to read product prices from a product file as items are scanned, and then the items file is updated with the items sold and how many. Such data is called persistent, because it is data that are retained after the program is completed and the machine is turned off.

Applications can store data persistent in several ways, and examples are

- data can be saved in ordinary "flat" files
- objects can be serialized to a file
- data can be saved in databases
- data can be saved as XML documents
- data can be saved as json documents

In this note I only will look at ordinary files, including serialization of objects. In fact, I already used files several times in the previous books, but the following is a more detailed review and a description of many of the classes associated with the IO.

Previously, the use of ordinary files to save data, were very common, and although it no longer has the same interest, there are still many situations where there is a need to store data in or read data from ordinary files. On the whole, there are many examples where it is interesting to be able to manipulate the file system from an application, and Java has an extensive API with classes for the treatment of files. Object serialization is also a matter of storing data in files, but in such a way that you can save an arbitrary object of any depth in a file – and load the object again. For many simple applications are object serialization and deserialization interesting, and in addition, serialization is used to send objects through a network.

A program with many transactions, and it is a program which very often must save data and read data from an external media will in practice always use a database. A database is maintained by a software package which include a daemon, that constantly is waiting for the other applications to enter a request for specific data items or to update the database. Use of databases from a Java program is the theme for the next book.

The classes to files are included in the package *java.io*, but since the package was developed, Java has expanded with a new package called *java.nio* – primarily for new needs and to support the facilities in modern operating systems. Both packages are used and will probably be that for many years to come, so the book covers in following both these two APIs.

The book consists basically of three parts, wherein the first two deals with IO and that the third part deals with the binary representation of data. If you do not have knowledge of binary and hexadecimal numbers, it might be a good idea first to read the books appendix.

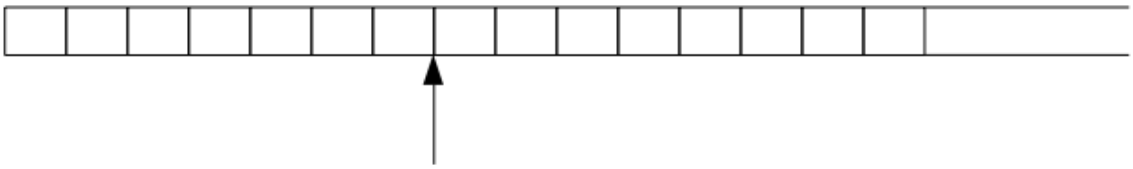
2 JAVA.IO

This chapter covers stream's, which is a collection of classes that provides the basic input and output operations in Java. A stream is a sequence of data that can be sent over a communication channel between a source and a destination. Both the source and destination may be several things as a program, file, one or another external device such as a printer, a network socket or an array in the computer's memory. Java provides a large number of classes available, all of which are intended to encapsulate the many details related to sending data over a stream, and it applies to both a stream of bytes, other primitive types and objects of all kinds. This chapter shows the use of many of the classes from *java.io*, and the classes are presented primarily through small test methods in the project *FileProgram*.

As mentioned above, the concept of a stream has many applications, but this book focus is primarily on streams between a file and a program.

2.1 FILES

You can think of a file as an infinite array with elements of the type *byte*:



where a file has a so-called file pointer represented by the arrow. The file pointer indicates where the next file operation starts, and the file pointer (the arrow) is moved forward corresponding to the number of bytes read or written. Files are stored on the machine's disk, and the operating system organizes the files in the form of file system that keeps track of directories and files. Each file has a unique name consisting of the name of the directory that contains the file, and the file name which must be unique within the directory. Java has a class called *File*, which represents a file by its name and provides a number of methods available that can be used to manipulate the file.

FILE INFO

The following method prints information about a file:

```
private static void fileInfo(String filename) throws IOException
{
  File file = new File(filename);
  System.out.println("Absolute path =  " + file.getAbsolutePath());
```

```
  System.out.println("Canonical path = " + file.getCanonicalPath());
  System.out.println("Name =           " + file.getName());
  System.out.println("Parent =         " + file.getParent());
  System.out.println("Path =           " + file.getPath());
  System.out.println("Is absolute =    " + file.isAbsolute());
  System.out.println(file.exists() ?
    "The file object exists" : "The file object is not found");
  if (file.exists()) System.out.println(file.isDirectory() ?
    "The file object is a folder" : "The file object is a file");
  System.out.println(file.length());
  System.out.println();
}
```

The method has a parameter that is the name of a file. Note that the method can raise an *IOException*, and it generally occurs, if you try to perform a method that works on a file, and the method of one reason or another can not be performed. In this case it is only one of the methods that can raise an *IOException*. The method creates a *File* object that represents a file named *filename*. You should note that a *File* object can represent both a directory and a file, and that it is not a requirement that the file (or the directory) exists, or is a legal file name. These conditions have first importance when the object is used. The first two statements prints the absolute file name and the canonical file name. In most cases it is the same, but the canonical file name is the simplets possible full file name and may be different than the absolute file name (see the test of the program).

The next three statements prints respectively the file name, the parent and the path. Here is the file name alone the name of the filen within the directory where it belongs, while the parent is the file name of the directory that the file is a part of. Path is the name used to create the *File* object and hence in this case the value of the parameter *filename*. The last three statements tests whether it is an absolute file name, and whether there exists a diretory, or a file with that name. In this case is tested whether it is a directory or a file. Finally the last statements prints the file size, measured in bytes.

The following test method uses the method *fileInfo()* to print information about various *File* objects:

```
private static void test01()
{
  System.out.println("Working Directory = " + System.getProperty("user.dir"));
  try
  {
    fileInfo("/home/pa/doc");
    fileInfo("/home/pa/doc/Swing.odt");
```

```
      fileInfo("/home/pa/doc/xpq/1234");
      fileInfo(".");
      fileInfo("..");
      fileInfo("doc");
    }
    catch (IOException ex)
    {
      System.out.println(ex);
    }
}
```

Note especially the first statement that prints the name on the current directory. You are invited to test the example and study the result.

DRIVE INFO

The following method has a *File* object as a parameter, and the method prints information about that disk partition, that contains the file:

```
private static void print(File root)
{
  System.out.println("Partition:      " + root);
```

```
    System.out.println("Free space =    " + root.getFreeSpace());
    System.out.println("Usable space = " + root.getUsableSpace());
    System.out.println("Total space =  " + root.getTotalSpace());
    System.out.println();
  }
```

All sizes are specified in bytes. When *getFreeSpace()* and *getUsableSpace()* usually do not show the same value, it is because the latter is a better estimate, since it takes into account the overhead associated with the current operating system's file system. The method is used in the following test method:

```
private static void test02()
{
  print(new File("/"));
  print(new File("/home"));
  print(new File("/run/media/pa/HD-PNFU3"));
}
```

which for my machine prints information about the *root* partition, the partition for *home* and for an external hard drive.

THE SIZE OF A FILE AND FILTERS

The following method has as parameter a file name and creates a corresponding *File* object. If this object represents a file, the method prints the name and the size where the size is determined recursively by the method *getSize()*, which determines the size of the file tree having the object file as a root. Note that *size* is a static variable, which is defined at the start of the program.

```
private static void fileSize(String filename, FileFilter filter)
{
  File file = new File(filename);
  if (file.exists())
  {
    System.out.println(file.getAbsolutePath());
    size = 0;
    getSize(file, filter);
    System.out.println(size);
  }
  else System.out.println("Filen findes ikke");
}
```

The method has a parameter whose type is *Filter*. It is an interface that defines a single method

```
public boolean accept(File pathname);
```

The method should test whether a file identified by the parameter meets certain criteria. The *FileFilter* object is tranfered to the method *getSize()* together with the *File* object that is represented by the file name:

```
private static void getSize(File file, FileFilter filter)
{
  if (file.isDirectory())
  {
    File[] files = filter == null ? file.listFiles() : file.listFiles(filter);
    for (File f : files) getSize(f, filter);
  }
  else if (filter == null || filter.accept(file)) size += file.length();
}
```

The method tests whether the first argument is a directory. Is this the case, the method *listFiles()* is called, which returns an array of all files in this directory. The method exists in several overloadings and include a method that as a parameter has a filter and returns only the files that meet the filter. With the array (as possibly can be empty) available the method *getSize()* is called recursively for all files in the array. If the method is called and the first parameter is not a directory, there is nothing else than the static variable *size* is counted with the file's size – if it is a file to be included according a possibly filter.

Below is a test method that determines the size of all files partly under the current directory and partly under the user's home directory. In both cases, it happens with and without a filter, where the filter accepts files that are either a directory or a regular file whose name ends with *.java*:

```
private static void test03()
{
  FileFilter filter = new FileFilter() { public boolean accept(File f) {
    return f.isDirectory() || (f.isFile() && f.getName().endsWith(".java")); } }
  fileSize(".", null);
  fileSize(".", filter);
  fileSize(System.getProperty("user.home"), null);
  fileSize(System.getProperty("user.home"), filter);
}
```

You must note how the filter is defined on the basis of an anonymous class.

MANIPULATION OF THE FILE SYSTEM

The class file also has a variety of methods that can manipulate the file system, for example create directories and files and delete them again. There are many (and you are encouraged to examine which), and the following test method shows the use of a few of them, but first a little helper method:

```
private static void showFiles(File file)
{
  System.out.printf(file.isDirectory() ? "[%s]\n" : "%s\n", file.getName());
  if (file.isDirectory())
  {
    File[] files = file.listFiles();
    for (File f : files) showFiles(f);
  }
}
```

The method prints recursive a file tree with a particular *File* object as root. The test method is presented below:

```
private static void test04()
{
```

```
  File file = new File(System.getProperty("user.home") + "/test");
  try
  {
    file.mkdir();
    (new File(file, "filer")).mkdir();
    (new File(file, "filer/file1")).createNewFile();
    (new File(file, "filer/file2")).createNewFile();
    showFiles(file);
    file.deleteOnExit();
  }
  catch (Exception ex)
  {
    System.out.println(ex);
  }
}
```

The first statement defines a *File* object that references a file named *test* in the user's home directory. Next, is created a directory with this name and including a sub-directory named *files*. Below again are created two ordinary files, then the method *showFiles()* is called. The result is the following:

```
[test]
[filer]
file2
file1
```

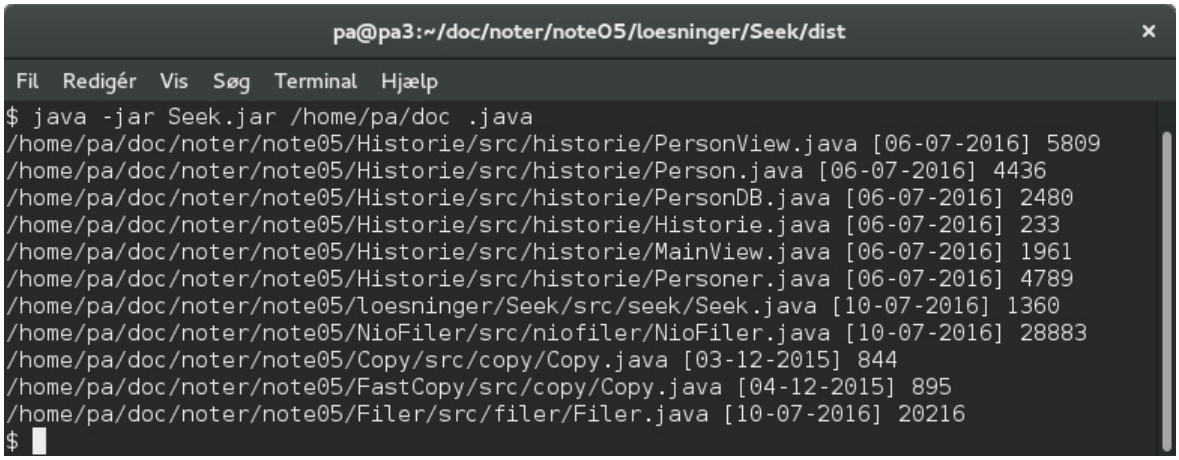
The last statement in the test method performs *deleteOnExit()*, which deletes all files and directories as this *File* object has created. The method has probably not a great practical use, but it is smart for testing to clean up as in this example.

EXERCISE 1

You must write a command, you can call *Seek*. The command shoul be performed in the following manner

```
java -jar Seek.jar directory [text]
```

where *directory* is the name of a directory, while *text* is a search text. The command should print the absolute file names of all files in the file tree whose root is *directory* and the file name contains the search text *text*. The result could be the following:



where for each file are shown the file name, the date when the file was last modified and the file size. The program must not display the names of directories that match the search criteria, and if the search criterion is empty (the last parameter is missing), the command should display all files in the current file tree.

If the first parameter is not the name of an existing directory, the command must simply print an error message on the screen.

2.2 RANDOM ACCESS FILES

Above I described the class *File*, and how a *File* object can be used to get information about files and directories and to manipulate the file system. In the rest of this chapter, I will show how to operate on the individual files and thus primarily how to write data to files and read the content of files. I'll start with *RandomAccessFiles*, which are files where you can both read and write data and thus files that programs can as use as simple databases. In fact, that kind of files has not so much interest today, but they are good to know, and to know how a program writes to and reads from files, and that is why I will start with random access files. They are in Java represented by the class *RandomAccessFile*. Consider the following test method:

```
private static void test05()
{
  int t1 = 0x41424344;
  int t2 = 0x45464748;
  int t3 = 0x494a4b4c;
  int t4 = 0x4d4e4f50;
  int t5 = 0x51525354;
  System.out.println(t1);
  System.out.println(t2);
  System.out.println(t3);
  System.out.println(t4);
```

```
System.out.println(t5);
RandomAccessFile dataFile = null;
try
{
  dataFile = new RandomAccessFile("numbers.dat", "rw");
  dataFile.writeInt(t1);
  dataFile.writeInt(t2);
  dataFile.writeInt(t3);
  dataFile.writeInt(t4);
}
catch (IOException ex)
{
  System.out.println(ex);
}
finally
{
  if (dataFile != null) try { dataFile.close(); } catch (IOException e) {}
}
try (RandomAccessFile file = new RandomAccessFile("numbers.dat", "r"))
{
  int sum = file.readInt();
  sum += file.readInt();
  sum += file.readInt();
  sum += file.readInt();
```

```
  System.out.println(t1 + t2 + t3 + t4);
  System.out.println(sum);
}
catch (IOException ex)
{
  System.out.println(ex);
}
try (RandomAccessFile file = new RandomAccessFile("numbers.dat", "rw"))
{
  file.seek(8);
  file.writeInt(t5);
}
catch (IOException ex)
{
  System.out.println(ex);
}
try (RandomAccessFile file = new RandomAccessFile("numbers.dat", "r"))
{
  long sum = file.readInt();
  sum += file.readInt();
  sum += file.readInt();
  sum += file.readInt();
  System.out.println((long)t1 + t2 + t5 + t4);
  System.out.println(sum);
}
catch (IOException ex)
{
  System.out.println(ex);
}
try (RandomAccessFile file = new RandomAccessFile("numbers.dat", "r"))
{
  byte[] arr = new byte[16];
  file.read(arr);
  String str = new String(arr);
  System.out.println(str);
}
catch (IOException ex)
{
  System.out.println(ex);
}
try (RandomAccessFile file = new RandomAccessFile("numbers.dat", "r"))
{
  double x1 = file.readDouble();
  double x2 = file.readDouble();
  System.out.println(x1);
  System.out.println(x2);
}
```

```
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

The first thing that happens is the creation of 5 *int* variables that are written on the screen:

```
1094861636
1162233672
1229605708
1296977744
1364349780
```

The numbers are not so interesting, but you should note how they are initialized as hexadecimal values consisting of 4 bytes. If you do not know hexadecimal numbers, you can read the appendix to this book.

To be able to write to a file, it must first be opened and must then be closed again. In this case is defined a variable *dataFile* of the type *RandomAccessFile*. It is opened in the try block with the statement

```
dataFile = new RandomAccessFile("numbers.dat", "rw");
```

The statement creates a file under the current directory with the name *numbers.dat*, while the last parameter tells how to open the file. Basically, there are two options *r* and *rw* (there are actually two other options). Here *rw* means you can both read and write to the file, while *r* means that you can only read the file. The above statement thus opens the file for both reading and writing. If the file does not exist, it is created, but otherwise it is just opened and the file pointer is placed at the beginning of the file, so the next file operation is carried out from this place.

After the file is opened, the first 4 of the above numbers are written to the file. The class *RandomAccessFile* has a method *writeInt()*, which writes an integer to the file. Exactly what happens is that the method writes 4 bytes from the location where the file pointer is, and after the method is performed, the file pointer is moved 4 bytes forward. The result is that the four *writeInt()* statements writes 16 bytes to the file, which represents the 4 integers.

After the four statements is performed, the file is closed. It is important, as the *writeInt()* statements writes to a memory buffer. The content of this buffer is first written physical to the file when the buffer is full or the file is closed. Another reason for the file to be closed is that you thereby releases the resources allocated to the file. Closing a file with the *close()* is important and to ensure that the statement is executed, it is placed in a *finally* block that is performed regardless of whether an exception occurs or not.

The above actually defines a pattern for how to treat files:

1. open the file
2. performs the wanted file operations
3. close the filen

After having written four numbers to the file, it is opened again to read the contents:

```
try (RandomAccessFile file = new RandomAccessFile("tal.dat", "r"))
```

This time the file is opened in a statement after the *try*. It is a syntax that is really just a short way of writing the above, but it guarantees that the file will be closed when the program exits the try block. The syntax is in most cases preferable, first, it is actually shorter and more readable, and secondly you are sure that you remember to close the file. After the file is opened, it performs a *readInt()* operation four times that reads four integers in the file, and the sum of these numbers is determined and printed along with the sum of the four numbers that are written in the file. The idea is that you have to see that you get the same result:

```
4783678760
4783678760
```

and thus it is the same four numbers that are read as the four numbers originally written in the file. You should note that the method *readInt()* reads 4 bytes from the location where the file pointer is and converts these 4 bytes to an *int*. After the method is performed, the file pointer is moved 4 bytes forward. The method interprets therefore not what the 4 bytes contains, and where the result is a "sense" integer. 4 bytes can be interpreted as an integer, no matter what they contains. As a next step the file is opened again for reading and writing:

```
try (RandomAccessFile file = new RandomAccessFile("tal.dat", "rw"))
{
  file.seek(8);
  file.writeInt(t5);
}
```

Then the file pointer is moved 8 bytes forward, which means that it points at the start of the third number in the file. The method then writes another number to the file, which means that the third number is overwritten, and the value is now the value of *t5*. After these statements are executed, the file is opened again for reading, and the sum of the four numbers determined again, and you will find that you get a different result, but the correct result:

```
4918422832
4918422832
```

In a *RandomAccessFile* you can thus use the method *seek()* to move the file pointer to a random position and read or write from that position.

The file is opened for reading again:

```
try (RandomAccessFile file = new RandomAccessFile("tal.dat", "r"))
{
  byte[] arr = new byte[16];
  file.read(arr);
  String str = new String(arr);
  System.out.println(str);
}
```

and the code creates an array with room for 16 bytes (that exactly matches the contents of the file – if you examine its properties with *Files* you can see that it takes up 16 bytes). Next, read the file with the method *read()* and the array as a parameter. This method reads a maximum of 16 bytes in the file (less if there are not 16 bytes after the file pointer). The 16 bytes are used to initialize a string, and the result is:

```
ABCDEFGHQRSTMNOP
```

The goal is to show that even though the file contains four integers which total takes up 16 bytes, it can be interpreted as anything, and it is the program that reads the file's contents that determines how the results should be interpreted. It is further illustrated with the last statements in the test method that reads the contents of the file and interprets it as two numbers of the type *double*.

HISTORY AGAIN

In the last example in the the book Java 2 I showed a program where you could enter information about historic people and save the information by object serialization. I will show a different version of the program where the difference is that the persons instead are stored in a *RandomAccessFile*. Let me say immediately that this solution has no advantages, and the goal is only to show an example of a file containing records of fixed length, formerly a typical application of files.

The starting point was the following class, which represents a historic person (where I have not shown the *get* and *set* methods):

```
public class Person implements Comparable<Person>, Serializable
{
  private int recnr = -1;    // idenfification of a person
  private String name;       // the person's name
  private String job;        // the person's position
  private String text;       // a description
  private int from;          // birth, start of reign or otherwise
  private int to;            // year of when the person is dead
```

```
public Person(String name, String job, String text, int from, int to)
{
   this.name = name;
   this.job = job;
   this.text = text;
   this.from = from;
   this.to = to;
}

public Person(int recnr, String navn, String job, String tekst, int fra, int til)
{
   this(navn, job, tekst, fra, til);
   this.recnr = recnr;
}
```

Compared to the previous version of the program I have made a single change, since the class is extended with an *int* variable named *recnr*. There are also defined both *get* and *set* methods for that variable, and that should contain a person's location in the file. Finally is added an extra constructor. The class is defined *Serializable*, and as *Person* objects no longer should be serialized, it could be deleted.

The file must contain objects of the type *Person*, but to be able to read these objects again (place the file pointer where a person starts), they must have a fixed size. Because the first three of the variables are strings, they do not have a fixed length. Therefore, it is necessary to select a maximum length of each of these values. As you will see in the next book, the same is necessary if *Person* objects should be stored in a database, so in fact the limitation is quite usual. The following class defines a file (a database) to *Person* objects or individual records:

```
package history;

import java.io.*;
import palib.util.*;
public class PersonDB
{
  public static final int NLENGTH = 50;
  public static final int JLENGTH = 30;
  public static final int TLENGTH = 500;

  private static final int RLENGTH = 2 * (NLENGTH + JLENGTH + TLENGTH) + 12;
  private RandomAccessFile file = null;

  public PersonDB(String path) throws IOException
  {
    file = new RandomAccessFile(path, "rw");
  }

  public void append(Person pers) throws IOException
  {
    pers.setRecnr(length());
    file.seek(file.length());
    write(pers);
  }

  public int length() throws IOException
  {
    return (int) file.length() / RLENGTH;
  }

  public Person select(int recnr) throws IOException
  {
    if (recnr < 0 || recnr >= length())
      throw new IOException("Illegal recordnumber");
    file.seek(recnr * RLENGTH);
    return read();
  }
```

```
public void update(Person pers) throws IOException
{
  if (pers.getRecnr() < 0 || pers.getRecnr() >= length())
    throw new IOException("Illegal record number");
  file.seek(pers.getRecnr() * RLENGTH);
  int rec = file.readInt();
  if (rec != pers.getRecnr()) throw new IOException("Illegal recordnumber");
  file.seek(pers.getRecnr() * RLENGTH);
  write(pers);
}

public void delete(Person pers) throws IOException
{
  if (pers.getRecnr() < 0 || pers.getRecnr() >= length())
    throw new IOException("Illegal recordnumber");
  file.seek(pers.getRecnr() * RLENGTH);
  file.writeInt(-1);
}

public void close()
{
  try
  {
    if (file != null) file.close();
  }
  catch (IOException ioe)
  {
  }
  file = null;
}

private void write(Person pers) throws IOException
{
  file.writeInt(pers.getRecnr());
  file.writeChars(Str.left(Str.cut(pers.getName(), NLENGTH), NLENGTH, ' '));
  file.writeChars(Str.left(Str.cut(pers.getJob(), JLENGTH), JLENGTH, ' '));
  file.writeChars(Str.left(Str.cut(pers.getText(), TLENGTH), TLENGTH, ' '));
  file.writeInt(pers.getFrom());
  file.writeInt(pers.getTo());
}

private Person read() throws IOException
{
  int rec = file.readInt();
  if (rec == -1) return null;
  String navn = read(NLENGTH);
```

```
    String job = read(JLENGTH);
    String tekst = read(TLENGTH);
    int fra = file.readInt();
    int til = file.readInt();
    return new Person(rec, navn, job, tekst, fra, til);
  }

  private String read(int length) throws IOException
  {
    StringBuilder buff = new StringBuilder();
    for (int i = 0; i < length; ++i) buff.append(file.readChar());
    return buff.toString().trim();
  }
}
```

The class starts with defining some constants that specifies that the name has of maximum of 50 characters, that the position has a maximum 30 characters, and the description a maximum of 500 characters. Finally is define a constant that indicates the size of a record. The sum of these three constants are multiplied by 2, because a character (a *char*) occupies 2 bytes. The last number 12 is the size of three integers. This means that a *Person* record always occupies 1172 bytes in the file. You should note that in most cases it will be considerably more than necessary and that the file will thus contain some wasted space, but these are the conditions for a file of this type. This is the price that you must pay to be able to read a record stored at a specific location.

The class has an instance variable of the type *RandomAccessFile* and the constructor opens the file for read and write. Note that the constructor raises an exception if the file can not be opened. The class offers the following services:

- *length(),* that returns the number of records in the file
- *select()*, that returns the record with a certain record number
- *append()*, that adds a new record to the file
- *update()*, that modify the content of a certain record
- *delete()*, that delete a certain record
- *close()*, that closes the file

The implementation of these methods is generally without problems, but *delete()* is an exception. The question is what should happens when you delete a record, as you logically get a "hole" in the file. In practice, there are several solutions. For example you could move the last record to the place where the deleted record was and then reduce the length of the file with the size of a record. The main problem with this strategy is, that a record in this way must change the record number, and the number then can not be used to identify a certaing record – the record number can not be the key. In this case, I will use a strategy where the record number set to -1, thus indicating that the record is deleted. The disadvantage of this approach is that the file that will contain a number of "dead" records, and you should therefore test whether a record is deleted when you reads in the file. Has deleted many records, it may also mean that the file size is unnecessarily large. The problem can be solved by reusing deleted records when adding new records (it is not done in this case). You must specifically noting, how the method *delete()* are implemented, and how to delete a record by writing the number -1 to the file. When the record number is the first field, it corresponds exactly to write the number -1 to the file.

The method *append()* have as a parameter, the *Person* object to be added to the file. The method starts by allocating the object's record number as the next record in the file, and then set the file pointer to the end of the file. The object is then written to the file with the method *write()*. Here you must notice how to save a *String* that happens with *writeChars()*. First are used two methods from *PaLib* that ensures that the string is not too long, and in the case where the *String* is shorter than maximum, it is filled with blanks to the maximum width.

The method *update()* works in principle in the same way, only is used the *Person* object's record number to position the file pointer in the right place. The method first reads an *int* to test whether it is the right record number, and if it is the case, writes the updates to the file.

The previous version of the program has a class called *Persons*, and it is rewritten as shown below, because the program instead of store *Person* objects by serilalization uses an object of type *PersonDB*. The class is expanded with a new instance variable called *path*, which is initialized by the constructor and contains the file name, but otherwise the class works in principle in the same way as in the first version of the program. The various methods are of course changed, but the difference is only that they store the objects in a different way. I've shown the whole class below, and you should study the code and how the class *PersonDB* is used:

```
package history;

import java.util.*;
import java.io.*;

public class Persons implements Iterable<Person>
{
  private static String path;
  private ArrayList<Person> list;

  public Persons(String path) throws IOException
  {
    this.path = path;
    load();
  }

  public Iterator<Person> iterator()
  {
    return list.iterator();
  }
```

```
public boolean add(Person pers)
{
   PersonDB db = null;
   try
   {
      db = new PersonDB(path);
      db.append(pers);
      list.add(pers);
      return true;
   }
   catch (IOException ex)
   {
      return false;
   }
   finally
   {
      if (db != null) db.close();
   }
}
```

```
public boolean remove(Person pers)
{
  PersonDB db = null;
  try
  {
    db = new PersonDB(path);
    db.delete(pers);
    return list.remove(pers);
  }
  catch (IOException ex)
  {
    return false;
  }
  finally
  {
    if (db != null) db.close();
  }
}

public boolean update(Person pers)
{
  PersonDB db = null;
  try
  {
    db = new PersonDB(path);
    db.update(pers);
    return true;
  }
  catch (IOException ex)
  {
    return false;
  }
  finally
  {
    if (db != null) db.close();
  }
}

private void initialize()
{
  list = new ArrayList();
  for (int i = 0;i < navne.length; ++i)
  {
    String job = navne[i][0].equals("Margrete d. 1.") ||
      navne[i][0].equals("Margrethe d. 2.") ? "Queen" : "King";
   int fra = navne[i][1].length() > 0 ? Integer.parseInt(navne[i][1]) : -9999;
    int til = navne[i][2].length() > 0 ? Integer.parseInt(navne[i][2]) : 9999;
   list.add(new Person(navne[i][0], job, "", fra, til));
```

```
    }
    store();
  }

  private void load() throws IOException
  {
    File file = new File(path);
    if (file.exists() && file.isFile())
    {
      PersonDB db = null;
      try
      {
        list = new ArrayList();
        db = new PersonDB(path);
        for (int rec = 0; rec < db.length(); ++rec)
        {
          Person pers = db.select(rec);
          if (pers != null) list.add(pers);
        }
      }
      finally
      {
        if (db != null) db.close();
      }
    }
    else initialize();
    Collections.sort(list);
  }

  private void store()
  {
    PersonDB db = null;
    try
    {
      db = new PersonDB(path);
      for (Person pers : list) db.append(pers);
    }
    catch (Exception ex)
    {
      System.out.println(ex);
    }
    finally
    {
      if (db != null) db.close();
    }
  }
```

```
  private static String[][] navne = {
    { "Gorm den Gamle", "", "958" },
    { "Harald Blåtand", "958", "986" },
    …
  };
}
```

Then there are the classes for the user interface, which I have not shown, and they are basically not changed compared to the first version of the program.

PROBLEM 1

You must in this problem write a program similar to the program above, but the program must instead maintain a product database. A product is described with four fields:

```
public class Product
{
  private int pnr;      // product number
  private String name;  // product name
  private short units;  // units in stock
  private float price;  // unit price
```

The product number is a sequential number starting with 0 (the first product should have product number 0), and each time a new product is created, it is assigned the next number, which thus corresponds to the product's record number in a *RandomAccessFile*. The name is a *String* that shall not exceed 100 characters. The units in stock must be a non-negative integer, and the unit price must be a positiv number (a *float*).

Start for example by writing the class *Product* finished. It is a simple class that should not contain much more than *get* and *set* methods for the four variables.

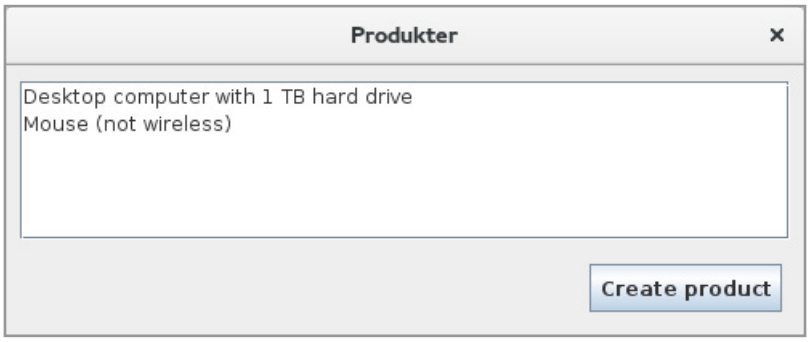
The program's model can be represented by the following class:

```
package productsprogram;

import java.io.*;
import java.util.*;
import palib.util.*;

/**
 * Class which represents a product database consisting of Product objects.
 * The first 4 bytes in the file is an integer that is interpreted as a signature
 * for the file.
 * When the file is opened, you can test this signature and thus get an indication
 * that the file is a product database.
 */
public class ProductDB
{
  private static final int SIGNATUR = 0x12345678;
  private static final int NAMESIZE = 100;
  private static final int RECORDSIZE = 2 * NAMESIZE + 10;

  private String path;

  /**
   * Constructor that initializes the file's path. The constructor does not open
   * the file, but validates the name and creates possible the file.
   * If the Name can not be properly validated, the construction raises an
   * exception. Following are validated
   *   If the file path eksistener is validated for the case of an ordinary file.
   *    If it this the case testes whether the file start with the correct
   *   signature, and is this not the case the constructor raises an exception.
   *   If the file path does not exist, the constructor creates a file with the
   *    correct signature.
   * @param path The files path
   * @throws IOException If the path could not be correct validated
   */
  public ProductDB(String path) throws IOException {}
```

```
  /**
   * Adds a new prooduct to the file. The method assigns the product a product
   * number that is the next record in the file. Thus the first product get product
   * number 0.
   * @param prod The product to be added to the file.
   * @throws IOException If the product could not be added
   */
  public void add(Product prod) throws IOException {}

  /**
   * Returns the product with product number pnr. If the product is not found, the
   * method raises an exception.
   * @param pnr The product number
   * @return The product with product number pnr
   * @throws IOException If the product is not found
   */
  public Product select(int pnr) throws IOException {}
  /**
   * Returns all products in the file.
   * @return AAll products in the file
   */
  public ArrayList<Product> select() {}
  /**
   * Updates a product. The method tests whether the record on that position has
   * the correct
   * product number. If not raised an exception.
   * @param prod The product to be updated
   * @throws IOException If the product is not found
   */
  public void update(Product prod) throws IOException {}

  /**
   * Deletes the product whose product number is pnr. The product is deleted by
   * writing the -i for product number.
   * @param pnr Product number for the product to be deleted
   * @throws IOException If the product is not found
   */
  public void delete(int pnr) throws IOException {}
}
```

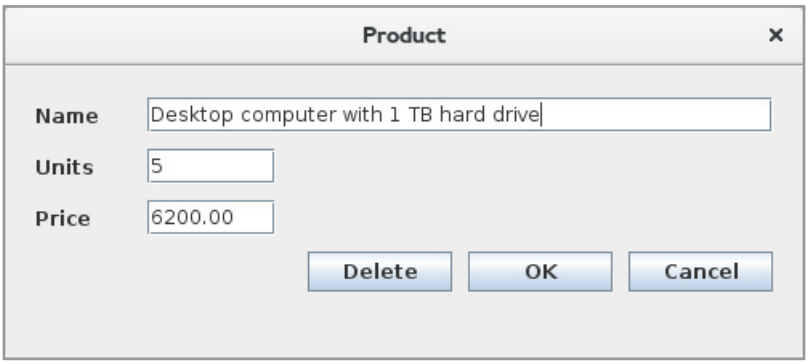
You should note that each of the class's methods should both open and close the file. You should also note that the first 4 bytes in the file is a signature, which you must take into account when determining the location of each record.

The application's main window should primarily display a list box with a list of all products:



Clicking on the button, you get a window where you can create a new product.

If you double-click on an product in the list box, you get the same window, where you can edit the product:



2.3 BYTE STREAMS

The package *java.io* contains several classes to byte streams and thus streams, which consists of a sequence of bytes, and basically there are the following classes:

- *ByteArrayInputStream* *ByteArrayOutputStream*
- *FileInputStream* *FileOutputStream*
- *ObjectInputStream* *ObjectOutputStream*
- *PipedInputStream* *PipedOutputStream*
- *FilterInputStream* *FilterOutputStream*

Here, the left column are input classes, which are all classes derived from an abstract class *InputStream*. The right column are classes for output, and is derived from an abstract class *OutputStream*. In the following I will show examples that use these classes – except *PipedInputStream* and *PipedOutputStream* that is postponed.

BYTEARRAYINPUTSTREAM AND *BYTEARRAYOUTPUTSTREAM*

One can think of a *ByteArrayOutputStream* as memory representation of an *OutputStream* and a *ByteArrayInputStream* as a memory representation of an *InputStream*. The typical use of these classes is to load the contents of a file (for example a picture) to a *ByteArrayOutputStream* where the file's content then can be manipulated in memory. Then the content is streamed to a file using a *ByteArrayInputStream*. Consider as an example, the following test method:

```
private static void test06()
{
  try (RandomAccessFile file = new RandomAccessFile("numbers1.dat", "rw"))
  {
    for (int i = 1; i <= 10; ++i) file.writeInt(i);
  }
```

```
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  ByteArrayOutputStream streamOut = new ByteArrayOutputStream();
  try (RandomAccessFile file = new RandomAccessFile("numbers1.dat", "r"))
  {
    for (int i = 0; i < file.length(); ++i) streamOut.write(file.readByte());
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  byte[] bytes = streamOut.toByteArray();
  for (int i = 0; i < bytes.length; ++i) if ((i + 2) % 4 == 0) bytes[i] = 1;
  ByteArrayInputStream streamIn = new ByteArrayInputStream(bytes);
  try (RandomAccessFile file = new RandomAccessFile("numbers1.dat", "rw"))
  {
    for (int b = streamIn.read(); b != -1; b = streamIn.read()) file.writeByte(b);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (RandomAccessFile file = new RandomAccessFile("numbers1.dat", "r"))
  {
    for (int i = 0, n = (int)(file.length() / 4); i < n; ++i)
      System.out.println(file.readInt());
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

The method starts by creating a *RandomAccessFile* with the numbers from 1 to 10. The hexadecimal content of the file is:

```
0x00000001
0x00000002
0x00000003
0x00000004
0x00000005
0x00000006
0x00000007
0x00000008
0x00000009
0x0000000a
```

and the fills 40 bytes. As the next step the content of the file is loaded and streamed to a *ByteArrayOutputStream* named *streamOut*. The result is that the file's content are now in memory as *streamOut*.

Then the content of *streamOut* is referenced with a *byte* array, and every fourth byte is modified:

```
byte[] bytes = streamOut.toByteArray();
for (int i = 0; i < bytes.length; ++i) if ((i + 2) % 4 == 0) bytes[i] = 1;
```

The result is that the 40 bytes representing 10 integers now are the numbers 257, 258, …, 266:

```
0x00000101
0x00000102
0x00000103
0x00000104
0x00000105
0x00000106
0x00000107
0x00000108
0x00000109
0x0000010a
```

After each byte is manipulated the arrray is encapsulated in a *ByteArrayInputStream*, and the result is written back to the file.

The above example of using a *ByteArrayOutputStream* and a *ByteArrayInputStream* is not typical and should only show the syntax. Later I will show examples, where these classes are important.

FILEINPUTSTREAM AND *FILEOUTPUTSTREAM*

Of the above classes to byte streams are *FileInputStream* and *FileOutputStream* probably the most important as it is classes that can stream bytes from and to a file. I will show as an example a method that encrypts the content of a file. The example is suitable to demonstrate how to manipulate the content of a file, but the algorithm can not be used for practical encryption of files, since it is easy to determine the key. The procedure is as follows:

A byte can have 256 different values and I starts with a *byte* array with 256 places and initializes the array with the values 0–255 (below written in hexadecimal):

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
....
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Then I performes a permutation of the array (corresponding to a number of swaps of pair of elements). The result could, for instance be:

```
00 01 32 03 04 05 38 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 30 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 5c 2d 2e 2f
1b 31 02 33 34 35 36 37 06 39 3a 3b 3c 3d 3e 3f
40 41 42 43 4e 45 46 47 48 49 4a 4b 4c 4d 44 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 2c 5d 5e 5f
....
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

The permuted array is then the encryption key. Consider the following integer:

```
0x021b2c30
```

that is the number 35335216. If we now must encrypt this number with the above key the principle is that each byte is used as an index to the key and replaced with what the key contains at that index:

```
0x32305c1b
```

which is the number 842030107, that is a completely different number. It's fine to be able to encrypt data, but it is not much worth unless you can also decrypt the data. If you has the key, it is simple. For each byte you can find its index in the key and the byte is replaced with the index:

```
0x021b2c30
```

and you are back to the number that was originally encrypted.

Consider the following test method:

```
private static void test07()
{
  String path0 = "text0";
  String path1 = "text1";
  String path2 = "text2";
  FileOutputStream streamOut = null;
  try
  {
    streamOut = new FileOutputStream(path0);
    String navn = "ABCDEFGHIJKLMNOPQRSTUVXYZÆØÅ";
    byte[] bytes = navn.getBytes();
    streamOut.write(bytes);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  finally
  {
    if (streamOut != null) try { streamOut.close(); } catch (Exception e) {}
  }
  printTekst(path0);
  int[] key = createKey();
  crypt(path0, path1, key);
  printTekst(path1);
  decrypt(path1, path2, key);
  printTekst(path2);
}
```

The first part of the method writes a string

```
String navn = "ABCDEFGHIJKLMNOPQRSTUVXYZÆØÅ";
```

to an *FileOutputStream*. You should note, how to create a *FileOutputStream*, converts the string to a *byte* array and then write that array to the file. The file is called *text0* and saved in the current directory. If you examine the file, you will see that it takes up 31 bytes, which is perhaps not what one would expect, but the reason is, that the *String* class's *getBytes()* method converts a string to a byte array using the default character encoding which is UTF8. The string takes up 28 characters that except for the last three are encoded as a single byte, while the last three each uses 2 bytes. Therefore the 25 + 6 = 31 bytes.

The program also has a method that reads the content of a stream (a file), interprets the content as text and prints the text on the screen:

```
private static void printTekst(String path)
{
  FileInputStream stream = null;
  try
  {
    stream = new FileInputStream(path);
```

```
    ByteArrayOutputStream bytes = new ByteArrayOutputStream();
    for (int b = stream.read(); b != -1; b = stream.read()) bytes.write(b);
    System.out.println(new String(bytes.toByteArray()));
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  finally
  {
    if (stream != null) try { stream.close(); } catch (Exception e) {}
  }
}
```

In this case, a *FileInputStream* is opened, and the file's content are read to a *ByteArrayOutputStream*. This stream is then converted to a string that is printed on the screen. The method is used in the test method *test07()* to print the file's content on the screen.

The rest of the test method then performs an encryption of the file's text to the file *text1* and prints the content of this file on the screen. The following method creates the key by swapping random elements in the array *key* 1000 times:

```
private static int[] createKey()
{
  int[] key = new int[256];
  for (int i = 0; i < key.length; ++i) key[i] = i;
  Random rand = new Random();
  for (int n = 0; n < 1000; ++n)
  {
    int i = rand.nextInt(key.length);
    int j = rand.nextInt(key.length);
    if (i != j)
    {
      int t = key[i];
      key[i] = key[j];
      key[j] = t;
    }
  }
  return key;
}
```

In principle, it is possible to create 256! keys that are a vastly large number, so it is a hopeless task to guess the key, but in return it is relatively easy to determine the key in plain text analysis.

The following class is used to encrypt a file:

```
class InCrypter extends FilterOutputStream
{
  private int[] key;

  public InCrypter(OutputStream stream, int[] key)
  {
    super(stream);
    this.key = key;
  }

  public void write(int b) throws IOException
  {
    out.write(key[b]);
  }
}
```

The class inherits *FilterOutputStream*, a stream that filters (manipulates) the bytes sent to the method *write()*, which collects the individual bytes in a buffer before being printed to an *OutputStream*. In this case, the *OutputStream* is sent as a parameter to the constructor together with the key, and the stream object is transferred to the base class. The *write()* method is known by the name *out*.

With this class the file can be encrypted in the following way:

```
private static void crypt(String path1, String path2, int[] key)
{
  FileInputStream in = null;
  InCrypter inCrypter = null;
  try
  {
    in = new FileInputStream(path1);
    inCrypter = new InCrypter(new FileOutputStream(path2), key);
    for (int b = in.read(); b != -1; b = in.read()) inCrypter.write(b);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  finally
  {
    if (in != null) try { in.close(); } catch (IOException e) {}
    if (inCrypter != null) try { inCrypter.close(); } catch (IOException e) {
  }
}
```

path1 is the name of the file to be encrypted, and it is opened as a *FileInputStream*. *path2* is the name of the file, that must contain the encrypted data and opens as a *FileOutputStream* used as a paramterer to an *InCrypter* object. After this happens the encryption by reading the input file and writes it to the output file using the *FilterOutputStream* object. You should note that when this stream is closed in the finally block, also the *OutputStream* is closed so the content is written physically to the file.

Decryption of the file takes place in principle in the same way. I first defines a *FilterInputStream* as a filter for an *InputStream*:

```
class DeCrypter extends FilterInputStream
{
  private int[] key;

  public DeCrypter(InputStream stream, int[] key)
  {
    super(stream);
    int[] map = new int[256];
    for (int i = 0; i < map.length; ++i) map[key[i]] = i;
    this.key = map;
  }
```

```
  public int read() throws IOException
  {
    int value = in.read();
    return (value == -1) ? -1 : key[value];
  }
}
```

The constructor has this time an *InputStream* as a parameter, but also the constructor inverts the key to make it easier to decrypt the file. The method *read()* reads the next byte in the input stream and is it a byte (and not EOF), the method returns the inverted key. The following method can be used to decrypt an encrypted file:

```
private static void decrypt(String path1, String path2, int[] key)
{
  DeCrypter deCrypter = null;
  FileOutputStream out = null;
  try
  {
    deCrypter = new DeCrypter(new FileInputStream(path1), key);
    out = new FileOutputStream(path2);
    for (int b = deCrypter.read(); b != -1; b = deCrypter.read()) out.write(b);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  finally
  {
    if (deCrypter != null) try { deCrypter.close(); } catch (IOException e) {}
    if (out != null) try { out.close(); } catch (IOException e) {}
  }
}
```

Here *path1* is the name of the file to be decrypted, and *path2* is the name of the decrypted file. The method works in principle in the the same manner as the method *crypt()*, only with the difference that it use a *FilterInputStream*.

If you run the program, you get the following result:

```
ABCDEFGHIJKLMNOPQRSTUVXYZÆØÅ
##[3z~g~
ABCDEFGHIJKLMNOPQRSTUVXYZÆØÅ
```

If you examines the three files using *Files* and you should note that all three files have the same size (31 bytes).

EXERCISE 2

Write a program named *Copy* that should be performed as a command and works in much the same way as the command *cp* in Linux, where the command line specifies the file – source – to be copied, and the name of the file – destination – as it should be copied to. The source file must be an *InputStream* and the destination must be an *OutputStream*.

When the program is finished, you must test it, and preferably with files of different types, such as a document and a picture.

DATA STREAM'S

Byte streams are the basic streams, and in principle, all streams are byte streams, but there are other classes that make it easy to stream other primitive types to a file. It is a *DataOutputStream* which is derived from *FilterOutputStream* and *DataInputStream* that is derived from the *FilterInputStream*. I will also mention a *BufferedOutputStream* and a *BufferedInputStream*, which also are derived from the classes *FilterOutputStream* and *FilterInputStream*. In general, the data streams works as an *OutputStream* and an *InputStream* in the sense that they calls the operating system for each byte to be written and for each byte to be reading. It is not particularly effective and, therefore the classes *BufferedOutputStream* and *BufferedInputStream*, that assign buffers to an *OutputStream* and an *InputStream*.

Consider the following test method:

```
private static void test08()
{
  try (DataOutputStream stream =
    new DataOutputStream(new BufferedOutputStream(new FileOutputStream("data"))))
  {
    stream.writeInt(123);
    stream.writeDouble(Math.PI);
    stream.writeUTF("Åge Sørensen");
    stream.writeLong(Long.MAX_VALUE);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (DataInputStream stream =
    new DataInputStream(new BufferedInputStream(new FileInputStream("data"))))
  {
    System.out.println(stream.readInt());
    System.out.println(stream.readDouble());
    System.out.println(stream.readUTF());
```

```
      System.out.println(stream.readLong());
    }
    catch (IOException ex)
    {
      System.out.println(ex);
    }
}
```

First the method opens a *FileOutputStream* called *data*. It is then encapsulated in a *BufferedOutputStream*, which in turn is encapsulated in a *DataOutputStream*. Then the method writes an *int*, a *double*, a *String* and a *long* to the file. You should note that a *DataOutputStream* has write methods for all the primitive types and also the type *String*. Here, the method *writeInt()* write 4 bytes to the file, while the method *writeDouble()* writes 8 bytes. The method *writeUTF()* writes a string converted with an UTF8 encoding. The string fills in this case, 12 characters, and because of the characters *Å* and *ø* it will therefore be encoded as 14 bytes, and the method should therefore write

4 + 8 + 14 + 8 = 34 bytes

to the file. If you examine the file, you will find that it takes up 36 bytes. The reason is that the method *writeUTF()* adds two additional bytes (the first two bytes), which indicates the number of bytes written to the file (the length of the string). It is necessary to read the string again.

The last part of the test method reads the content of the file again, and here you must note that a *DataInputStream* has read-methods for all the primitive data types. For example the method *readDouble()* that reads the next eight bytes from the location where the file pointer is and convert the bytes to a *double*. Similarly, *readUTF()* to reads the first two bytes, and from these determines how many bytes to be read.

EXERCISE 3

You must write a program that has a method that prints 1000000 random numbers of the type *double* to a file. The numbers should be written using a *DataOutputStream*, but without using a *BufferedOutputStream*. When the method writes data to the file, it must at the same time determines the sum of the numbers and print the value on the screen.

The program must also have a method that reads the numbers in the above file and determines their sum. The numbers must be read using a *DataInputStream*, but without using a *BufferedInputStream*. Finally, the method must print the sum on the screen and it would like to be the same number as in the first method.

Finally, try to change the above methods, so the first uses a *BufferedOutputStream* and the other uses a *BufferedInputStream*. Try to observe an increased efficiency. If not, try to increase the number of numbers to 10000000.

EXERCISE 4

You have to write the same program as in exercise 2, but this time you must encapsulate the two streams respectively in a *BufferedOutputStream* and a *BufferedInputStream*.

Can you observe an improvement?

2.4 OBJECT SERIALIZATION

In the examples in the previous books I have used serialization of objects as an easy way to store data in a file. I will now take a closer look at how it works. In principle it is a matter of stream an object's bytes to a *ByteStream*, but it is not necessarily simple. If you consider a *DataStream*, it is simple because for the primitive types, you know how much a datalement fills, and then exactly how many bytes to be written or read. This is expressed by the classes *DataOutputStream* and *DataInputStream* that has specific methods for each of the primitive data types. When you look at a *String* (which is an object) it is necessary with a little more since it is necessary also to save, how many bytes the string fills. It is necessary to read the string again. The situation is even more complex if there is any object, since one does not know what such an object fills, and when it may be composed of primitive types and objects that in turn can consist of primitive types and objects, and it may repeat at an arbitrary depth.

In order to load an object from a file, it is necessary, together with the object's data to store information about the object's structure, and as an object, in principle, is a hierarchical structure there must be stored information about the entire structure, such that the object can be restored when it is deserialized. It is not free, to serialize and deserialize objects and is not necessarily effective and is done using a technique called *reflection*, which are explained in a recent book. For this reason, Java supports three types of serialization, which is illustrated in the following examples.

In order that an object can be serialized, it must implement the interface *Serializable*. It is a simple interface without fields or methods. If you try to serialize an object whose class is not *Serializable* you get an *Exception*. As an example is shown a class that is defined Serializable:

```
class Person implements Serializable
{
  private String name;
  private Calendar fdate;
  private double sats;
  private transient int hours = 0;

  public Person(String name, Calendar fdate, double sats)
  {
    this.name = name;
    this.fdate = fdate;
    this.sats = sats;
  }
```

```
public String getName()
{
   return name;
}

public int getAge()
{
   Calendar dato = Calendar.getInstance();
   int age = dato.get(Calendar.YEAR) - fdate.get(Calendar.YEAR);
   if ((dato.get(Calendar.MONTH) + 1) * 100 + dato.get(Calendar.DATE) >=
     (fdate.get(Calendar.MONTH) + 1) * 100 + fdate.get(Calendar.DATE)) ++age;
   return age;
}

public double getPay()
{
   return hours * sats;
}

public void add(int hours)
{
   this.hours += hours;
}
```

```
  public String toString()
  {
    return name + String.format(", %d years", getAge());
  }
}
```

The class has four instance variables. It can be serialized as the first variable has the type *String*, which is *Serializable*, the other is of the type *Calendar* (and must represent a person's date of birth) is also *Serializable*, while the third is a *double* (and all variables of a primitive types can immediately be serialized). The variable will represent an hourly rate. The last variable represents the number of hours, and it is defined *transient*. Such a variable is not serialized, and the meaning is that you then can specify that variables or objects of any reason not should be serialized.

The following method creates a *Person* object, assigns the person 20 hours, and then prints the object. Next, the object is serialized to a file. An object is serialized with an *ObjectOutputStream* (which as parameter has an *OutputStream*) and the method *writeObject()*:

```
private static void test09()
{
  try (ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream("person")))
  {
    Person pers =
      new Person("Carlo Jensen", new GregorianCalendar(1949, 0, 23), 225);
    pers.add(20);
    print(pers);
    stream.writeObject(pers);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream("person")))
  {
    Person pers = (Person)stream.readObject();
    print(pers);
  }
  catch (ClassNotFoundException ex)
  {
    System.out.println(ex);
  }
```

```
    catch (IOException ex)
    {
      System.out.println(ex);
    }
  }
```

After the object is serialized it is deserialized again with an *ObjectInputStream*, after which it is printed on the screen. The object is read with *readObject()*, and you should note the need for a typecast. If it not is possible, you get a *ClassNotFoundException*. The *print()* method is:

```
private static void print(Person pers)
{
  System.out.println(pers);
  System.out.println("Pay: " + pers.getPay());
}
```

and if the test method is performed, you get the result:

```
Carlo Jensen, 68 years
Pay: 4500.0
Carlo Jensen, 68 years
Pay: 0.0
```

You should note that the value of the variable *hours* is not serialized.

SERIALIZES MULTIPLE OBJECTS

The following method also serialize *Person* objects (three objects), and should show that it is possible to serialize multiple objects in the same file:

```
private static void test10()
{
  Person[] pers = {
    new Person("Carlo Jensen", new GregorianCalendar(1949, 0, 23), 225),
    new Person("Gudrun Andersen", new GregorianCalendar(1963, 10, 3), 325),
    new Person("Abelone Sørensen", new GregorianCalendar(1972, 3, 13), 375) }
  try (ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream("personer")))
  {
    for (Person p : pers) stream.writeObject(p);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
```

```
  try (ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream("personer")))
  {
    for (;;)
    {
      Person p = (Person)stream.readObject();
      System.out.println(p);
    }
  }
  catch (EOFException ex)
  {
  }
  catch (ClassNotFoundException ex)
  {
    System.out.println(ex);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

However, it is rare that you do that. If you have multiple objects and as here an array of objects (or an *ArrayList* of objects), you will usually serialize it all but as a single operation:

```
private static void test11()
{
  try (ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream("personer")))
  {
    Person[] pers = {
      new Person("Carlo Jensen", new GregorianCalendar(1949, 0, 23), 225),
      new Person("Gudrun Andersen", new GregorianCalendar(1963, 10, 3), 325),
      new Person("Abelone Sørensen", new GregorianCalendar(1972, 3, 13), 375) };
    stream.writeObject(pers);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream("personer")))
  {
    Person[] pers = (Person[])stream.readObject();
    for (Person p : pers) System.out.println(p);
  }
  catch (ClassNotFoundException ex)
  {
    System.out.println(ex);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

It is possible, as the class *Array* is *Serializable*. The same applies to the collection classes, and therefore they can also be serialized.

SERIALIZES AN OBJECT HIERARCHY

The following class inherits *Person* and expands the class *Person* with a job title

```
class Employee extends Person
{
  private String job;
```

```
  public Employee(String name, String job, Calendar fdate, double sats)
  {
    super(name, fdate, sats);
    this.job = job;
  }

  public String toString()
  {
    return job + "\n" + super.toString();
  }
}
```

You should note that the class is not defined *Serializable*, but objects of this type can still be serialized because the base class is *Serializable*:

```
private static void test12()
{
  try (ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream("employee")))
  {
    Employee e = new Employee("Carlo Jensen", "Executioner",
      new GregorianCalendar(1949, 0, 23), 225);
    stream.writeObject(e);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream("employee")))
  {
    Employee e = (Employee)stream.readObject();
    System.out.println(e);
  }
  catch (ClassNotFoundException ex)
  {
    System.out.println(ex);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

That is, it works because an *Employee* is *Serializable* as it inherits a *Serializable* class.

USER DEFINED *WRITEOBJECT()* AND *READOBJECT()*

Sometimes you want to determine by yourself how the objects must be serialized and deserialized and depart from the default operations. A typical reason could be performance, but there could also be other reasons. The process is quite simple, since the class it is desired to serialize as usual must be defined *Serializable* and in addition implements the methods *writeObject()* and the *readObject()*. Then these methods are used instead of the default methods in *ObjectOutputStream* and *ObjectInputStream*. The following class implements these methods:

```
class Name implements Serializable
{
  private String name;
  private Calendar time = null;

  public Name(String name)
  {
    this.name = name;
  }
```

```
  private void writeObject(ObjectOutputStream out) throws IOException
  {
    out.writeUTF(name);
  }

  private void readObject(ObjectInputStream in)
    throws ClassNotFoundException, IOException
  {
    name = in.readUTF();
    time = Calendar.getInstance();
  }
  public String toString()
  {
    return name + "\n" + (time == null ? "Not desirialized" : getTime());
  }

  private String getTime()
  {
    return
      String.format("%02d-%02d-%04d %02d:%02d:%02d:%03d", time.get(Calendar.DATE),
       time.get(Calendar.MONTH) + 1, time.get(Calendar.YEAR),
       time.get(Calendar.HOUR_OF_DAY), time.get(Calendar.MINUTE),
       time.get(Calendar.SECOND), time.get(Calendar.MILLISECOND));
  }
}
```

When an object is serialized, only the field name will be saved while the date first is created when the object is deserialized and thus shows when the object is deserialized:

```
private static void test13()
{
  try (ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream("name")))
  {
    Name name = new Name("Carlo Jensen");
    stream.writeObject(name);
    System.out.println(name);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream("name")))
  {
    Name name = (Name)stream.readObject();
    System.out.println(name);
  }
```

```
    catch (ClassNotFoundException ex)
    {
      System.out.println(ex);
    }
    catch (IOException ex)
    {
      System.out.println(ex);
    }
}

Carlo Jensen
Not desirialized
Carlo Jensen
26-12-2016 12:14:51:792
```

You should note that the serialization and deserialization syntactically happens in exactly the same way. It's just some other methods that are performed.

EXTERNALIZABLE

It is also possible completely self to address how the objects should be serialized and deserialized, and again could performance be a reason. The class that should be serialized must instead of *Serializable* implement the interface *Externalizable*, which defines two methods *writeExternal()* and *readExternal()*:

```
class King implements Externalizable
{
  private String name;
  private int from;
  private int to;

  public King()
  {
  }

  public King(String name, int from, int to)
  {
    this.name = name;
    this.from = from;
    this.to = to;
  }
```

```
  public String toString()
  {
    if (from == Integer.MIN_VALUE && to == Integer.MIN_VALUE) return name;
    if (to == Integer.MIN_VALUE) return name + String.format("  : %d -", from);
    if (from == Integer.MIN_VALUE) return name + String.format("  : - %d", to);
    return name + String.format("  : %d - %d", from, to);
  }
  public void writeExternal(ObjectOutput out) throws IOException
  {
    out.writeUTF(name);
    out.writeInt(from);
    out.writeInt(to);
  }

  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException
  {
    name = in.readUTF();
    from = in.readInt();
    to = in.readInt();
  }
}
```

These methods must then respectively write to an *ObjectOutput* and reading from an *ObjectInput*, there are the interfaces, as are implemented by an *ObjectOutputStream* and an *ObjectInputStream*. The syntax for serialization and deserialization of an object is in turn the same as before:

```
private static void test14()
{
  try (ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream("king")))
  {
    King king = new King("Knud den Hellige", 1080, 1086);
    stream.writeObject(king);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream("king")))
  {
    King king = (King)stream.readObject();
    System.out.println(king);
  }
  catch (ClassNotFoundException ex)
  {
    System.out.println(ex);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

2.5 CHARACTER STREAMS

There also is a family of classes that are used to stream a sequence of characters and thus a text. In fact, they are far more often needed than directly stream a sequence of bytes (and hence the raw data). In Java is text or characters represented as 2 bytes unicodes, but when they are streamed to a file, a character fill one or a few bytes as text is encoded as UTF8. All the regular letters occupies only one byte, while others that are not so frequently used characters, including national characters occupies 2 or more bytes. The goal is simple to reduce the number of bytes sent over a stream. Stream classes to characters must handle the necessary conversion that happens transparently without the programmer's involvement – at least as long that the program should not be used anywhere in this world. Strictly speaking, not the same encoding are used everywhere but the entire Western world uses the same encoding. If you need internationalization, it is possible to specify the encoding to be used, but otherwise stream classes uses an encoding based on the machine's local setting. Below is an example that writes text to a file and read the text again:

```
private static void test15()
{
  try (FileWriter writer = new FileWriter("names"))
  {
    writer.write("Gorm den Gamle\n");
    writer.write("Harald Blåtand\n");
    writer.write("Svend Tveskæg\n");
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  char[] buffer = new char[20];
  try (FileReader reader = new FileReader("names"))
  {
    for (int count = reader.read(buffer, 0, buffer.length); count != -1;
      count = reader.read(buffer, 0, buffer.length))
        for (int i = 0; i < count; ++i) System.out.print(buffer[i]);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

The methods writes strings separated by line breaks, and the text are loaded again in a buffer whose content then are printed on the screen. When you in this way wants to deal with text files, you will usually encapsulate both the writer and the reader in a buffer, and here you should especially notice how to read files with the method *readLine()* and thus perceive the file as line-oriented:

```
private static void test16()
{
  try (BufferedWriter writer = new BufferedWriter(new FileWriter("names")))
  {
    writer.write("Gorm den Gamle");
    writer.newLine();
    writer.write("Harald Blåtand");
    writer.newLine();
    writer.write("Svend Tveskæg");
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (BufferedReader reader = new BufferedReader(new FileReader("names")))
  {
```

```
    for (String line = reader.readLine(); line != null; line = reader.readLine())
      System.out.println(line);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

In the above example I used a *FileWriter* and *FileReader* as character-oriented stream classes to files and including *BufferedWriter* and *BufferedReader*, but there are other *Reader* and *Writer* classes, there most be important:

- *FileReader*	*FileWriter*
- *FilterReader*	*FilterWriter*
- *StringReader*	*StringWriter*
- *PipedReader*	*PipeWriter*
- *CharArrayReader*	*CharArrayWriter*
- *InputStreamReader*	*OutputStreamWriter*
-	*PrintWriter*

You should also note that a *FileReader* and *FileWriter* is wrapper classes for respectively an *InputStreamReader* and an *OutputStreamWriter*.

Sometimes it is recommended not to use the *FileReader* and *FileWriter*, as they do not allow you to enter any *Encoding*. The same method as above can be written as follows where you indicates the encoding:

```
private static void test17()
{
  try (BufferedWriter writer = new BufferedWriter(
    new OutputStreamWriter(new FileOutputStream("navne2"), "ISO-8859-1")))
  {
    writer.write("Gorm den Gamle");
    writer.newLine();
    writer.write("Harald Blåtand");
    writer.newLine();
    writer.write("Svend Tveskæg");
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
```

```
  try (BufferedReader reader =
    new BufferedReader(new InputStreamReader(
      new FileInputStream("navne2"), "ISO-8859-1")))
  {
    for (String line = reader.readLine(); line != null; line = reader.readLine())
      System.out.println(line);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

In this case, indicates the encoding that it not should be UTF8 but ISO 8859-1, in which all characters are encoded as a single byte. The main use of this option is that you may need to read text files created with another program and then not may be UTF8 encoded.

PROBLEM 2

The folder to this book contains three text files called

1. *regions*, that contains a line for each Danish region where a line consists of a region number and the name of the region separated by commas
2. *municipalities*, that contains a line for each Danish municipality, where the line consists of the municipality's number, the name og the municipality and the number of the region that the municipality belongs and where the three fields are separated by commas
3. *zipcodes*, that contains a line for each zip code, where the line consists of the postal code and the city name, followed by one or more municipality numbers indicating the municipalities that use this zip code and where all fields are separated by commas

You must now create a new project in NetBeans, as you for example can call for *Denmark*. You must then add the following three interfaces to your project and write classes that implements these interfaces:

```
package denmark;
/**
 * Interface, thar defines a region.
 * Two regions are equal if they have the same region number.
 * Regions are ordred ascending after name.
 * The iterator pattern must be implemented to iterates this
 * region's municipalities.
 */
```

```
public interface IRegion extends
  Comparable<IRegion>, Iterable<IMunicipality>, java.io.Serializable
{
  /**
   * @return The region's number
   */
  public int getRnr();

  /**
   * @return The regions name
   */
  public String getName();

  /**
   * @return Number of municipalities in this region
   */
  public int getSize();

  /**
   * Returns the municipality in this region with number mnr
   * @param mnr Municipality number
   * @return The municipality in this region with number mnr
   * @throws Exception If the region not has a municipality with number mnr
   */
```

```
  public IMunicipality getMunicipality(int mnr) throws Exception;

  /**
   * Change the region's name
   * @param name The regions name
   */
  public void setName(String name);

  /**
   * Add a municipality to this region.
   * @param municipality The municipality to be added
   * @throws Exception If the region already has a municipality with that number
   */
  public void addMunicipality(IMunicipality municipality) throws Exception;

  /**
   * Remove a municipality from this region.
   * @param mnr Number of the municipality to be removed
   * @return True, if the municipality is removed
   */
  public boolean removeMunicipality(int mnr);
}

package denmark;

/**
 * Interface, that defines a municipality.
 * Two municipalities are considered equal if they have the same
 * municipality number.
 * Municipalities are ordered ascending by municipality name.
 * Iterator mønsteret definerer at man kan iterere over denne kommunes postnumre.
 */
public interface IMunicipality extends
  Comparable<IMunicipality>, Iterable<IZipcode>, java.io.Serializable
{
  /**
   * @return The municipality's number
   */
  public int getMnr();

  /**
   * @return The municipality's name
   */
  public String getName();
```

```
  /**
   * @return The region in which this municipality belongs
   */
  public IRegion getRegion();

  /**
   * @return Number of zip codes, as this municipality uses
   */
  public int getSize();

  /**
   * Returns the zip code with the number code if this municipality uses
   * this zip code.
   * @param code The code of the zip code to be returned
   * @return The zip code with the number code if this municipality uses this code.
   * @throws Exception If the municipality does not use this code
   */
  public IZipcode getZipcode(String code) throws Exception;

  /**
   * Changes the name of the municipality
   * @param name The new name
   */
  public void setName(String name);

  /**
   * Move this municipality in another region.
   * @param region The other region
   */
  public void setRegion(IRegion region);

  /**
   * Adds a zip code to this municipality, thus indicating a zip code that
   * this municipality uses.
   * @param zipcode The zip code that should be added
   * @throws Exception If this municipality already uses this code
   */
  public void addZipcode(IZipcode zipcode) throws Exception;

  /**
   * Removing a zip code from this municipality, thus indicating that
   * this municipality does not use the code.
   * @param code The zip code to be removed
   * @return True, if the zip code was removed
   */
  public boolean removeZipcode(String code);
}
```

```
package denmark;
/**
 * Interface, that defines a zip code.
 * Two zip codes are considered equal if they have the same number.
 * Zip codes are ordered ascending by the code.
 * the iterator pattern defines that you can iterates the municipalities, that
 * use this zip code.
 */
public interface IZipcode extends
  Comparable<IZipcode>, Iterable<IMunicipality>, java.io.Serializable
{
  /**
   * @return The code
   */
  public String getCode();

  /**
   * @return The city
   */
  public String getCity();
```

```
  /**
   * @return Number of municipalities that uses this zip code.
   */
  public int getSize();

  /**
   * Returns the municipality with number mnr if the municipality uses
   * this zip code.
   * @param mnr Municipality number
   * @return The municipality with number mnr if the municipality uses this code.
   * @throws Exception If the municipality with number mnr uses this zip code
   */
  public IMunicipality getMunicipality(int mnr) throws Exception;

  /**
   * Changed the zip code's city name
   * @param city City name
   */
  public void setCity(String city);

  /**
   * Adds a municipality to this Zipcode object and thus indicates that this
   * zip code is used by the municipality.
   * @param municipality The municipality to be added
   * @throws Exception If the municipality already added
   */
  public void addMunicipality(IMunicipality municipality) throws Exception;

  /**
   * Removes the municipality with the number mnr from this zip code and
   * thus indicates that the municipality does not use this zip code.
   * @param mnr The number of the municipality to be removed
   * @return True, if the municipality was removed
   */
  public boolean removeMunicipality(int mnr);
}
```

You must then write the following class, when the class must be written as a singleton:

```
package denmark;

import java.util.*;
import java.io.*;
```

```
public class Repository implements Serializable
{
  private static String filename = "denmark";    // filename to serialization the data
  private static Repository instance = null;     // variable to an instance
  private List<IRegion> regions;                 // to regions
  private List<IMunicipality> municipalities;    // to municipalities
  private List<IZipcode> zipcodes;               // to zip codes

  public static Repository getInstance()
  {
  }

  /**
   * Returns the zip code from the code.
   * @param code The code
   * @return The zip code if it is found and else null
   */
  public IZipcode getZipcode(String code)
  {
  }

  /**
   * Returns the region with region number rnr.
   * @param rnr Region number
   * @return The region if it is found and else null
   */
  public IRegion getRegion(int rnr)
  {
  }

  /**
   * Returns the municipality with number mnr.
   * @param mnr Municipality number
   * @return The municipality if it is found and else null
   */
  public IMunicipality getMunicipality(int mnr)
  {
  }

  /**
   * @return Iterator, that iterates til zip codes.
   */
  public Iterator<IZipcode> itrZipcode()
  {
  }
```

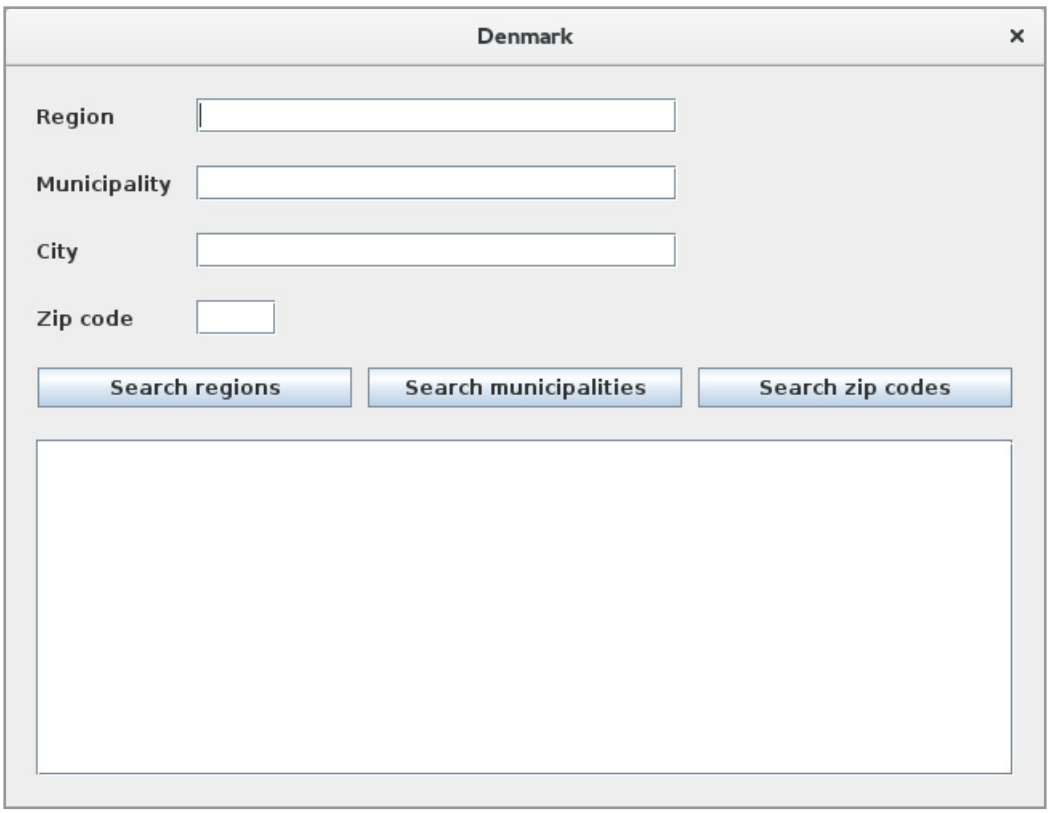
```
    /**
     * @return Iterator, that iterates til regions.
     */
    public Iterator<IRegion> itrRegioner()
    {
    }

    /**
     * @return Iterator, that iterates til municipalities.
     */
    public Iterator<IMunicipality> itrMunicipality()
    {
    }

    private Repository()
    {
    }
}
```

The class is defined *Serializable*, and the *private* constructor must start by deserialize an object of the type *Repository*. Is it not possible, the constructor instead must initialize an object from the contents of the three text files and then serialize the object.

The program should open the window, as shown below where it should be possible to search for regions, municipalities and zip codes. You can enter the following search texts:

1. *Region* that matches all regions where the name contains the search text
2. *Municipality*, that matches all municipalities where the name contains with the search text
3. *City* that matches all zip codes where the city starts with the search text
4. *Zip code* that maches all zip codes where the code starts with the search text

It is a part of the task to decide what it will mean if a search is combined of several criteria, but if, for example you searches municipalities and typed something for municipality and city, it might for example mean that you should see the names of all the municipalities where the municipality's name contains the search text for municipality and the municipality uses the zip code that matches the search text for city name.



2.6 TEXT SCANNER

During data entry for console applications I have used the class *Scanner*, and it can be used for anything other than simply entering data. Basicly a *Scanner* is an object that scans a character-oriented stream and divides the stream into tokens. Consider as an example the following method:

```
private static void test18()
{
  try (BufferedWriter writer = new BufferedWriter(new FileWriter("names")))
  {
    writer.write("Gorm den Gamle");
    writer.newLine();
    writer.write("Harald Blåtand");
    writer.newLine();
    writer.write("Svend Tveskæg");
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (Scanner scan = new Scanner(new BufferedReader(new FileReader("names"))))
  {
//       scan.useDelimiter("\n");
    while (scan.hasNext()) System.out.println(scan.next());
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

The method starts by creating a text file called *names*. Then the method writes three lines to the file, as are names of three Danish kings. The last part of the method creates a scanner, which scans the file *names* and then prints the tokens, that is determined, with one token at each line. The result is:

```
Gorm
den
Gamle
Harald
Blåtand
Svend
Tveskæg
```

By default tokens are separated by a white spaces, which are space, tab and newline. Therefore, the file *names* has 7 tokens. If you wish, you can specify how the tokens must be separated, and it you remove the comment in the above method, only line breaks are used to separates tokens, and the result is:

```
Gorm den Gamle
Harald Blåtand
Svend Tveskæg
```

The parameter to the method *useDelimiter()* is a string that can be any regular expression, and you is thus able to provide very complex patterns for seperation of tokens.

The class *Scanner* also has methods that can convert tokens to the primitive types – if the tokens has legal values. Otherwise the conversion throws an exception. Consider as an example the following method:

```
private static void test19()
{
   try (BufferedWriter writer = new BufferedWriter(new FileWriter("numbers")))
   {
      Random rand = new Random();
```

```
    for (int i = 0; i < 10; ++i)
      writer.write(String.format("%d %1.2f\n",
        rand.nextInt(10) + 1, rand.nextDouble() * 100));
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  try (Scanner scan = new Scanner(new BufferedReader(new FileReader("numbers"))))
  {
    double sum = 0;
    while (scan.hasNext()) sum += scan.nextInt() * scan.nextDouble();
    System.out.println(sum);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

The method prints 10 lines to a file in which each line contains an integer and a decimal number separated by a space. Next, the file is scanned, but as part of the scan the elements are converted (the individual tokens) to either an *int* or a *double* and the product is calculated.

EXERCISE 5

A common use of text files is to transfer data between applications. An application can store information in a text file that can be sent to a recipient who then read the content of the file. Often that kind of files are called CSV files. CSV stands for comma separated values, noting that the file contains values that are separated by a comma. The separation character do not necessarily needs to be a comma and can be anything. The only requirement is that it is a character that can not occur in the individual data elements (values). Besides comma are often used spaces, semicolons, tabulator, or equivalent, but whatever character you use the file is still called a CSV file. In this exercise you have to write a program that creates a CSV file and read it again.

Create a new project, you can call *CsvFiles*. Add a method that creates a text file which consists of lines that start with a date and is followed by one or more decimal numbers separated by semicolons. The start of the file could, for example be:

```
17.04.2014;573,33;228,97;242,14;394,81;190,60;412,17;107,02;249,49;293,29
11.09.2012;325,04;87,59;325,25;465,42
31.12.2013;992,33;695,32
```

A line could, for example be interpreted as a number of amounts (for example product sales) as concerning a certain date. You should note, that there can be several lines with the same date and in any order. The difficult thing is not to create the file, but to creates the lines to be printed to the file, so you should create some helper methods.

You must then write a method that reads the content of the file, and prints a list as shown below:

```
02.01.2012:              1848,44
04.01.2012:              3303,82
05.01.2012:               756,04
08.01.2012:              9773,91
```

The table must show the total sales for each date, and the lines must be sorted by date.

CSV files are typically line oriented, but it need not be the case. You have to write another method that creates a text file, but the file must consist of elements separated by semikolen which is alternately a date and a number. Next, write a method that reads the file and prints an overview as above. Because the file is not line oriented, it must be read in a different way, and this can advantageously be done by use of a scanner.

3 JAVA.NIO

The above defines in principle what is necessary in order to work with files, but modern operating systems provide many services available that are not supported by the classes in java.io, and it is the reason for another package called *java.nio*. Another reason for a new package for io is also providing better services available to stream data over a network. In this book I will only look at the new package for files, but in the book about network programming, I treat other services that *java.nio* provides. This chapter describes the main classes in this package. Basically it is about classes to

- buffers
- chanels
- paths

The fundamental concept in the new API's are buffers. A process as JVM performs IO by asking the operating system to drain the content of a buffer with a *write* operation, and accordingly, the JVM ask the operating system to fill a buffer with a *read* operation. If the JVM for instance ask the operating system to read from a disk file, the operating system will send a command to the disk controller to read a block of bytes from the disk and save them in an operating system buffer. After this operation is completed, the operating system copies the content of the buffer to a second buffer in the address space of the specified process as a parameter of the JVM in the *read()* operation. One of the goals with *java.nio* is to increase the efficiency of the necessary copying of data between buffers, and it is not simple, particularly because the disk controller always reads/writes with a fixed block size, while the JVM uses variable sized blocks and even blocks, which are not necessarily contiguous allocated and can be moved by the garbage collector.

One can think of a *channel* as a conduit used by the operating system to drain or fill a buffer with bytes from a device such as a disk controller. An example is a *FileChannel*, and as its name suggests it acts as a channel between a program and a disk file. It supports several things that are not possible with standard IO, and here include a locking mechanisms and memory mapped files.

Finally are *paths java.nio*'s representation of the file system.

The following describes the most important classes in the package *java.nio*, and it is primarily using simple test methods in the program *NioProgram*.

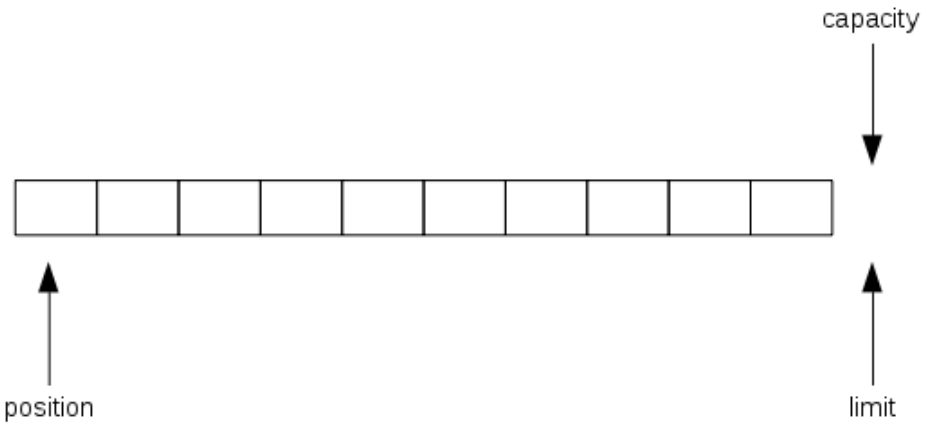
3.1 BUFFERS

A buffer is an object that can contain a fixed amount of data that can be sent to or received from an IO-service, and one can think of a buffer as an object that sits between a program and a channel. The buffer has basically four characteristics:

1. *capacity*, that defines how many data elements a *Buffer* can contain
2. *limit*, that is a 0-based index, that denotes the first element, that can not be written or read
3. *position*, that is a 0-based index, that denotes the position for the next read or write
4. *mark*, that is a 0-based index, that is used to reset the *position*

Buffer is an abstract class that provides a number of methods available, but there are concrete classes for each of the primitive data types, and basically a buffer is an encapsulation of an array.

You can think of a new created buffer as illustrated in the following figure:



The buffer has a *capacity* of 10, and the *limit* has always the same value as *capacity* and thus the value 10. The *position* is initially 0, indicating the element in the buffer, which can be accessed. *position* will always have a value between 0 and *limit*. The pointer *mark* is initially undefined. As an example is shown a method that creates a buffer of bytes, which has space for 10 elements:

```
private static void test01()
{
  Buffer buffer = ByteBuffer.allocate(10);
  print(buffer);
  buffer.limit(8);
  buffer.position(3);
  print(buffer);
  buffer.flip();
  print(buffer);
  buffer.mark();
  buffer.position(2);
  print(buffer);
  buffer.reset();
  print(buffer);
  buffer.limit(8);
  buffer.position(3);
  print(buffer);
  buffer.rewind();
  print(buffer);
  buffer.clear();
  print(buffer);
  buffer = LongBuffer.allocate(10);
  print(buffer);
}
```

```
private static Buffer print(Buffer buffer)
{
  System.out.println("Capacitet:  " + buffer.capacity());
  System.out.println("Limit:      " + buffer.limit());
  System.out.println("Position:   " + buffer.position());
  System.out.println("Remaining:  " + buffer.remaining());
  System.out.println(buffer);
  System.out.println();
  return buffer;
}
```

The first statement creates the buffer and when the method is performed is the result of the first print statement:

```
Capacitet: 10
Limit:     10
Position:  0
Remaining: 10
java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
```

Here you can see that *limit* from start is set to *capacity*, and the *position* is 0. You can also see that the buffer has 10 places available. Next *limit* is set to 8 and *position* to 3 and the second print statement prints the result

```
Capacitet: 10
Limit:      8
Position:   3
Remaining: 5
java.nio.HeapByteBuffer[pos=3 lim=8 cap=10]
```

You must specifically note that there are now 5 places available, and thus only access to elements with an index less than *limit* (the elements with index 3, 4, 5, 6 and 7). The next statement performs a *flip()*, which means that *limit* is set to *position*, while *position* is set to 0 (and a possible *mark* to undefined). The result of the third print statement is:

```
Capacitet: 10
Limit:      3
Position:   0
Remaining: 3
java.nio.HeapByteBuffer[pos=0 lim=3 cap=10]
```

The buffer has then free space for 3 elements. The next statement performs a *mark()*, which means that the value of the index *mark* is set to the value of *position*. Next position is set to 2 and the buffer is reprinted:

```
Capacitet: 10
Limit:      3
Position:   2
Remaining: 1
java.nio.HeapByteBuffer[pos=2 lim=3 cap=10]
```

where there is nothing to explain, but *mark* has the value 3. The next statement is a *reset()*:

```
Capacitet: 10
Limit:      3
Position:   0
Remaining: 3
java.nio.HeapByteBuffer[pos=0 lim=3 cap=10]
```

The method *reset()* sets limit to *mark* when *position* is set to 0, and then *mark* is undefined. After that *limit* and *position* are again assigned the values 8 and 3:

```
Capacitet: 10
Limit:     8
Position:  3
Remaining: 5
java.nio.HeapByteBuffer[pos=3 lim=8 cap=10]
```

after a *rewind()* is performing. It is a method that set *position* to 0, and any *mark* is then undefined:

```
Capacitet: 10
Limit:     8
Position:  0
Remaining: 8
java.nio.HeapByteBuffer[pos=0 lim=8 cap=10]
```

The next statement performs a *clear()*, which briefly means that all pointers are set back to start, and thus as they were immediately after the buffer was created:

```
Capacitet: 10
Limit:     10
Position:  0
Remaining: 10
java.nio.HeapByteBuffer[pos=0 lim=10 cap=10]
```

The last two statements creates a new buffer, but this time a buffer to values of the type *long*:

```
Capacitet: 10
Limit:     10
Position:  0
Remaining: 10
java.nio.HeapLongBuffer[pos=0 lim=10 cap=10]
```

As can be seen from the example, the type *Buffer* has a number of methods, which among other things can be used to manipulate the *Buffer's* pointers. These methods all return a *Buffer* object (a reference to the *Buffer* after the operation is performed), and this means that, instead of writing

```
buffer.mark();
buffer.position(2);
buffer.reset();
```

you can write

```
buffer.mark().position(2).reset();
```

The example shown has two types of buffers

- *ByteBuffer*
- *LongBuffer*

Both are examples of abstract classes and concrete buffers are created using static methods. There are also the following buffer classes:

- *CharBuffer*
- *ShortBuffer*
- *IntBuffer*
- *FloatBuffer*
- *DoubleBuffer*

CREATE BUFFERS

There are several ways to create a buffer, and the following method presents four options:

```
private static void test02()
{
  byte[] bytes = new byte[500];
  print(ByteBuffer.allocate(100));
  print(ByteBuffer.allocateDirect(100));
  print(ByteBuffer.wrap(bytes));
  print(ByteBuffer.wrap(bytes, 10, 300));
}
```

Generally, a buffer is an encapsulation of an array, and wee talk, where appropriate, of a *backing array*, but it need not be the case. The above method defines an array to be used as backing array to a buffer. The most common way to create a buffer, however, is directly to apply the method *allocate()* as in the first example above:

```
Capacitet: 100
Limit:     100
Position:  0
Remaining: 100
java.nio.HeapByteBuffer[pos=0 lim=100 cap=100]
```

There is not much to add in relation to the first example and the result is a buffer having a capacity of 100 bytes, and wherein the limit is equal to the capacity and the position is 0. The next buffer is almost identical:

```
Capacitet: 100
Limit:     100
Position:  0
Remaining: 100
java.nio.DirectByteBuffer[pos=0 lim=100 cap=100]
```

but you should note that the types are different, where the type of the first is *HeapByteBuffer*, while the type of the last is *DirectByteBuffer*. The difference is explained later. That I am talking about buffers to bytes, is not so important, and the syntax would be the same if there instead were talking about buffers to elements of the other primitive types.

The third buffer encapsulates the array of bytes:

```
Capacitet: 500
Limit:     500
Position:  0
Remaining: 500
java.nio.HeapByteBuffer[pos=0 lim=500 cap=500]
```

and the result is a buffer with a capacity of 500 elements, a limit of 500 and a position of 0. You should note that the capacity is determined by the length of the array, and the type is the same as in the first example. The last buffer is also created as an encapsulation of the array of bytes, but here you can set a position and a length which means that the limit will be *position* + *length*:

```
Capacitet: 500
Limit:     310
Position:  10
Remaining: 300
java.nio.HeapByteBuffer[pos=10 lim=310 cap=500]
```

Buffers created with *allocate()* and *wrap()* has a backing array, and you can access this array with the method *array()*.

You can also create a so-called *view buffer* that is a buffer that encapsulates a second buffer. The two buffers have the same data and have the same capacity, but they each have their limit, position and mark, and a view buffer is simply a matter of being able to manipulate the same data (the same buffer) in several ways. Consider the following method:

```
private static void test03()
{
  byte[] bytes = { 0x02, 0x03, 0x05, 0x07, 0x0b, 0x0d, 0x11, 0x13 };
  ByteBuffer buffer1 = ByteBuffer.wrap(bytes);
  ByteBuffer buffer2 = buffer1.duplicate();
  IntBuffer buffer3 = buffer1.asIntBuffer();
  buffer1.position(4);
  buffer1.limit(6);
  print(buffer1);
  print(buffer2);
  print(buffer3);
}
```

The method defines a byte array with 8 elements used to create a *ByteBuffer* named *buffer1*. For this object is used the method *duplicate()* to create a view buffer named *buffer2*. Next is created another view buffer named *buffer3*, but this time with the method *asIntBuffer()*. This buffer allows to manipulate the original buffer as was it an *IntBuffer*. As a next step the position and limit for buffer1 are set, after which the buffer it is printed:

```
Capacitet: 8
Limit:     6
Position:  4
Remaining: 2
```

There is not much mystery in the result, but if you subsequently prints *buffer2*, you will see that it has a different position and another limit, and are not changed when the *buffer1* changes its values:

```
Capacitet: 8
Limit:     8
Position:  0
Remaining: 8
java.nio.HeapByteBuffer[pos=0 lim=8 cap=8]
```

since a view buffer has its own indexes. If you prints the last buffer, you get the result:

```
Capacitet: 2
Limit:     2
Position:  0
Remaining: 2
java.nio.ByteBufferAsIntBufferB[pos=0 lim=2 cap=2]
```

Its capacity is 2. The array has elements for 8 bytes, which corresponds to 2 ints, and it corresponds precisely to that this view interprets the underlying array as an *int* array.

READ AND WRITE

Buffers are of course only interesting if you can modify the content, and it means that you can read the individual elements and change them. This is done with *put()* and *get()*, and both methods are in an absolute and a relative version. Consider the following method:

```
private static void test04()
{
  byte[] bytes = { 0x02, 0x03, 0x05, 0x07, 0x0b, 0x0d, 0x11, 0x13 };
  ByteBuffer buffer = ByteBuffer.wrap(bytes);
  printElements(buffer);
  buffer.put(3, (byte)23);
  buffer.position(6);
  buffer.put((byte)29);
  bytes[7] = 0x1f;
  printElements(buffer);
}

private static void printElements(ByteBuffer buffer)
{
  for (int i = 0; i < buffer.limit(); ++i) System.out.print(buffer.get(i) + " ")
  System.out.println();
  for (buffer.position(0); buffer.position() < buffer.limit(); )
    System.out.print(buffer.get() + " ");
  System.out.println();
}
```

The method *test04()* creates a buffer which encapsulates an array of 8 elements. Next the method prints the content of the buffer in two ways. The first is a regular for *loop*, where the individual components are referred to by the method *get()* and an index. The individual elements are referred thus by an absolute index, and the index must be greater than or equal to 0 and less than the limit. The next loop references the elements relative to there position, and the loop starts with setting the position to 0, and then the loop is iterated as long as the position is less than the limit. The elements are refered with *get()* without an index and the method returns the element to what position is pointing to, and then the position is counted up by 1. This means that when the loop stops, the position is equal to limit. The result is

```
2 3 5 7 11 13 17 19
2 3 5 7 11 13 17 19
```

After the content of the buffer are printed the elements with index 3 is changed to 23. It is by the method *put()* but with an absolute index. You should note that this operation does not change the value of *position*. Next the position is set to 6 and the value at position 6 is changed to 29. It also happens with *put()*, but without specifying an index and thus relative to the position, and after the operation is performed, the position is counted forward with 1. Finally the value of the last place in the backing the array is directly changed to 31, and is the buffer printed the result is:

```
2 3 5 23 11 13 29 31
2 3 5 23 11 13 29 31
```

get() and *put()* may also have an array as a parameter, and it can be used to, respectively, get more elements out of a buffer and modifying the multiple elements in a buffer more efficiently. Consider the following method:

```
private static void test05()
{
  byte[] bytes1 = { 0x02, 0x03, 0x05, 0x07, 0x0b, 0x0d, 0x11, 0x13 };
  byte[] bytes2 = { 0x17, 0x1d, 0x1f, 0x25, 0x2b };
  ByteBuffer buffer = ByteBuffer.allocate(8);
  printElements(buffer);
  buffer.position(0);
  buffer.put(bytes1);
  printElements(buffer);
  buffer.position(2);
  buffer.put(bytes2, 1, 3);
  printElements(buffer);
  byte[] bytes3 = new byte[4];
  byte[] bytes4 = new byte[8];
  buffer.position(2);
  buffer.get(bytes3);
  buffer.position(2);
  buffer.get(bytes4, 3, 3);
  printElements(ByteBuffer.wrap(bytes3));
  printElements(ByteBuffer.wrap(bytes4));
}
```

This method creates a buffer with room for 8 elements and prints the contents:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

That is, in a buffer created with *allocate()* elements are all 0. Next the position is set to 0 and the buffer is changed by means of the array *bytes1*:

```
buffer.put(bytes1);
```

This means that the contents of the array are copied to the buffer and from the starting position onwards. Therefore, it is necessary to first set position to 0, and after the statement is completed, the position has shifted eight places forward and thus to the end of the buffer. When you then prints the buffer, you get

```
2 3 5 7 11 13 17 19
2 3 5 7 11 13 17 19
```

There is another version of the *put()*, which in addition to the array indicates an offset into the array and the number of elements to be copied:

```
buffer.put(bytes2, 1, 3);
```

The position was previously set to 2, and the result is

```
2 3 29 31 37 13 17 19
2 3 29 31 37 13 17 19
```

where the places with index 2, 3 and 4 are updated. There are similar overloadings of *get()*, where the elements in the buffer is copied to an array. *bytes3* is an array with room for 4 elements, and the position is set 2, and the following statement copies the 4 elements to the array *bytes3*:

```
buffer.get(bytes3);
```

Thus, the array length determines how many elements are copied. If *bytes3* is encapsulated in a buffer and printed you gets

```
29 31 37 13
29 31 37 13
```

As the last example is *bytes4* an array with room for 8 elements. Position is again set to 2 and are the follwing statement performed

```
buffer.get(bytes4, 3, 3);
```

the elements at positions 3, 4 and 5 are copied:

```
0 0 0 29 31 37 0 0
0 0 0 29 31 37 0 0
```

A typical use of a buffer is a process (a program) which takes up elements of a buffer, and wherein the buffer is then connected to a chanel, which drain (taking elements from) the buffer. The following example defines an array with four strings and a *CharBuffer*. Then the method iterates over the strings and for each string the buffer is emptied – the method *clear()* that sets the limit to the buffer's capacity, and position to 0. Next, an inner loop iterates over each character in the string and fills the characters into the buffer. When the characters are loaded into the buffer, the buffer is drained by the method *printLine()*, which means that the content must be printed from the position 0 to the limit. That is, the limit should be set to the position, after which the position is to be set to 0 and this can be done with the method of *flip()*:

```
private static void test06()
{
  String[] names = { "Erik Ejegod", "Erik Emune",
"Erik Lam", "Erik Plovpenning" };
  CharBuffer buffer = CharBuffer.allocate(100);
  for (int i = 0; i < names.length; ++i)
  {
    buffer.clear();
    for (int j = 0; j < names[i].length(); j++) buffer.put(names[i].charAt(j));
    buffer.flip();
    printLine(buffer);
  }
}

private static void printLine(CharBuffer buffer)
{
  while (buffer.hasRemaining()) System.out.print(buffer.get());
  System.out.println();
}
```

Note especially how the method *printLine()* drains the buffer and how to use *hasRemaining()* to test whether the position is reached to limit. The method *test06()* is a kind of prototype of using a buffer.

Sometimes you is interested to drain a buffer and then subsequently put the indices back as they were before. It is here that the index *mark* can help. Consider the following method:

```
private static void test07()
{
  IntBuffer buffer = IntBuffer.allocate(10);
  buffer.put(2).put(3).put(5).put(7).put(11).put(13).put(17).put(19);
  buffer.limit(8).position(2).mark().position(6);
  print(buffer);
  buffer.reset();
  print(buffer);
}

private static void print(IntBuffer buffer)
{
  while (buffer.hasRemaining()) System.out.print(buffer.get() + " ");
  System.out.println();
}
```

This method creates an *int* buffer with room for 10 numbers. Next, 8 numbers are added to the buffer. Note the syntax, where the use of the method *put()* returns a reference to the buffer after the element has been added. After the 8 numbers are added the limit is set to 8, position 2, and then set a *mark* in order to finally set position to 6. Then the following show the indexes:

```
- limit = 8
- position = 6
- mark = 2
```

Then drain the buffer with the method *print()*, and the result is:

```
17 19
```

After the method *print()* is executed, the following applies regarding the indices:

```
- limit = 8
- position = 8
- mark = 2
```

The method *test07()* now performs a *reset()* and the result of the indexes are

```
- limit = 8
- position = 2
- mark = undefined
```

and the method *print()* will finally result in

```
5 7 11 13 17 19
```

As the last for read and write I will mention the method *compact()*, which copies all elements between the position and limit to start of the buffer. After that, the position is equal to the *limit – position*, while the limit is equal to capacity. Consider the following method:

```
private static void test08()
{
  IntBuffer in = IntBuffer.allocate(10);
  in.put(2).put(3).put(5).put(7).put(11).put(13).put(17).put(19).put(23).
    put(29).rewind();
  IntBuffer out = IntBuffer.allocate(10);
  while (in.hasRemaining())
  {
    out.put(in.get());
    if (out.position() == 8 && out.get(0) == 2)
    {
      out.flip().position(2);
      out.compact();
    }
  }
  print(out);
  out.clear();
  print(out);
}
```

If you performs the method, you get the result

```
0 0
5 7 11 13 17 19 23 29 0 0
```

which is not clear. The method starts by creating an *IntBuffer in* with room for 10 elements, after which the buffer is filled with 10 numbers. After the buffer is filled is the position is equal to 10, and therefore the initialization is finishes with a *rewind()*, which sets the position to 0. The next step is to create another *IntBuffer out* with room for 10 numbers. The following *while* loop drains the buffer *in* and fill the elements of the buffer *out*. When position in *out* is 8 (and the first element of the buffer is not 2), the following is the case concerning the indices:

```
- limit = 10
- position = 8
```

Once the position has been reached 8 the method performs a *flip()*, followed by a *position(2)*, and the indices are

```
- limit = 8
- position = 2
```

At that time, the content of the buffer is

```
2 3 5 7 11 13 17 19
```

and then there is performed a *compact()* to copy the elements from the index 2 through 7 to the start of the buffer:

```
5 7 11 13 17 19 17 19
```

and the indices which now applies

```
- limit = 10
- position = 8 - 2 = 6
```

The *while* loop adds two more elmenter to *out* (the numbers 23 and 29), which is on the places with index 6 and 7 and for the indices apply

```
- limit = 10
- position = 8
```

The first *print()* outside the loop therefore prints:

```
0 0
```

Before the last *print()* is performed, a *clear()* is executed that sets the position to 0 and the limit to capacity, and the result is therefore

```
5 7 11 13 17 19 23 29 0 0
```

LAST REMARKS ABOUT BUFFERS

It is possible to compare two buffers with *equals()*, and the protocol is as follows:

Two buffers are *equals()* if and only if they have the same element type, they have the same number of remaining elements, and the two sequences of remaining elements, considered independently of their starting positions, are individually equal.

As an example, the following method prints *true*:

```
private static void test09()
{
   IntBuffer buff1 = IntBuffer.allocate(10);
   buff1.put(0). put(1).put(2).put(3).put(5).put(7).put(11).put(13).put(17).put(19);
   buff1.position(2);
   buff1.limit(6);
```

```
    IntBuffer buff2 = IntBuffer.allocate(8);
    buff2.put(2).put(3).put(5).put(7);
    buff2.flip();
    System.out.println(buff1.equals(buff2));
}
```

First notice that the two buffers are of different lengths. *buff1* has position 2 and limit 6 while *buff2* have position 0 and limit 4, but it means that both buffers have four remaining elements and these elements are equal in pairs.

The last observation relates to performance. Currently, a buffer is nothing but a memory structure that encapsulates an array, and in relation to files is the meaning of course that the IO system must copy data to and from the buffers. Somewhere it is not optimal, since the operating system's IO operations are using their own buffers that do not have anything with Java to do, but buffers as described in this section are data structures for JVM, and therefore it will be advantageous that they were the same buffers. It is, however, not quite simple, in particular due to the Java's buffers are not necessarily contiguous allocated, but also because the garbage collector if necessary are moving them. In an attempt to solve these problems, one can create a byte buffer as

```
ByteBuffer buffer = ByteBuffer.allocateDiect(99);
```

It is a buffer that is associated with the IO and files, and in some cases, results in an improved in performance and defines a buffer without a backing array, but there must be special reasons for using that kind of buffers, since the gain is typically small.

3.2 CHANNELS

Channels belongs to buffers to achieve high-performance I/O. A channel is an object that represents an open connection to a hardware device as a file. Channels efficiently transfer data between byte buffers and operating system-based I/O services, and often there exists a one-to-one correspondence between a file descriptor and a channel. When you work with channels in a file context, the channel will often be connected to an open file.

A channel is defined by an interface

```
java.nio.channels.Channel
```

The interface only defines two methods *close()* and *isOpen()*, and the methods for IO are defined in two subinterfaces

```
java.nio.channels.WritableByteChannel
java.nio.channels.ReadableByteChannel
```

there as the names says defines channels for writing or reading. A class that only implements one of this interfaces is called *unidirectional*, and a class the implements both interfaces is called *bidirectional*.

There are several ways to create a channel to a file and thus a *Channel* object. The class *Channels* has two static methods:

```
WritableByteChannel writer = Channels.newChannel(OutputStream stream)
ReadableByteChannel reader = Channels.newChannel(InputStream stream)
```

Finally, several of the classes in the *java.io* are extended with a method *getChannel()*, which returns a *Channel*.

I will now show an example that uses two *Channel* objects to copy a file. The example assumes that my home directory has a subdirectory *data* with a file named *municipalities*, which has a size of 2863 bytes (the text file with the names of Danish municipalities, but is not important what the file contains, as long as it is found in the right directory and has a size between 2 and 3 KB). Consider first the following method:

```
private static void copy1(ReadableByteChannel src, WritableByteChannel dst)
  throws IOException
{
  ByteBuffer buffer = ByteBuffer.allocate(1024);
  while (src.read(buffer) != -1)
  {
    buffer.flip();
    dst.write(buffer);
    buffer.compact();
  }
  buffer.flip();
  dst.write(buffer);
}
```

The method has two parameters that are both unidirectional *Channel* objects, wherein the first one is for reading and the second one is for writing. The method creates a buffer, and the only thing you should notice is, that the size is smaller than the file. Next, the method performs a loop that iterates so long as there is data on the input channel. Each iteration performs a *flip()* of the buffer and then writing the buffer to the output channel as the method *write()* drains the buffer. You must note that it is all bytes from the position 0 to the limit and after the operation the position is equal to limit. Next, is performed a *compact()* statement, which copies depending bytes after the limit to the start of the buffer. This is to ensure that the buffer does not contain data, which is not drained to the output channel. After the loop stops – all data on the input channel is read – the method performs again a *flip()* and the buffer is drained to the output channel.

The following method creates an input channel to an *InputStream* and an output channel to an *OutputStream* and use the above method to copy a file:

```
private static void test10()
{
  ReadableByteChannel src = null;
  WritableByteChannel dst = null;
  try
  {
```

```
    src = Channels.newChannel(new FileInputStream(
      System.getProperty("user.home") + "/data/municipalities"));
    dst = Channels.newChannel(new FileOutputStream("municipalities1"));
    copy1(src, dst);
  }
  catch (IOException ex)
  {
    System.out.println("I/O error: " + ex.getMessage());
  }
  finally
  {
    try
    {
      if (src != null) src.close();
      if (dst != null) dst.close();
    }
    catch (IOException ex)
    {
      System.out.println("I/O error: " + ex.getMessage());
    }
  }
}
```

There is not much to explain, but you must note how to create the two *Channel* objects.

Below is another example on a method that copies a file by means of two *Channel* objects:

```
private static void copy2(ReadableByteChannel src, WritableByteChannel dst)
  throws IOException
{
  ByteBuffer buffer = ByteBuffer.allocate(4096);
  while (src.read(buffer) != -1);
  buffer.flip();
  dst.write(buffer);
}
```

The difference is that this time the entire input file is copied to the buffer before it is drained to the output channel. This method is simpler than the previous one, but there is an important condition, that the buffer has enough space to contain all the input file. The test method is as follows

```
private static void test11()
{
  try (ReadableByteChannel src =
    Channels.newChannel(new FileInputStream(
      System.getProperty("user.home") + "/data/municipalities"));
```

```
      WritableByteChannel dst =
        Channels.newChannel(new FileOutputStream("municipalities2")))
    {
      copy2(src, dst);
    }
    catch (IOException ex)
    {
      System.out.println("I/O error: " + ex.getMessage());
    }
  }
```

and here is the main difference, that the two *Channel* objects are created as part of the *try* statement.

MULTIPLE BUFFERS

As shown in the above examples is a *Channel* a fundamental technique to stream data between two end points, that for example can be files, and as an adapter between the channel and the program there is a buffer. For many years one of the most important ways to increase your computer's performence has been parallelism, a technique that particular operating system exploits and it also applies in connection with IO. Among other things for this reason *Channel* objects supports the use of multiple buffers where an input channel can fill several buffers, while an output channel can drain multiple buffers. The following example will show the syntax:

```
public static void test12()
{
  try (
    ScatteringByteChannel src = (ScatteringByteChannel) Channels.newChannel
      new FileInputStream(System.getProperty("user.home") + "/data/regions"));
    WritableByteChannel out = Channels.newChannel(System.out);
    GatheringByteChannel dst =
      (GatheringByteChannel) Channels.newChannel(new FileOutputStream("regions1"))
  {
    ByteBuffer[] buffers =
      { ByteBuffer.allocateDirect(100), ByteBuffer.allocateDirect(50) };
    src.read(buffers);
    for (int i = 0; i < buffers.length; ++i)
    {
      buffers[i].flip();
      out.write(buffers[i]);
      System.out.println("\n");
      buffers[i].rewind();
    }
```

```
      dst.write(buffers);
    }
    catch (IOException ex)
    {
      System.out.println("I/O error: " + ex.getMessage());
    }
}
```

After *try* are created three channels. An input channel, that uses multiple buffers, and is called a *scattering* channel. It is created as other input channels, and therefore it is necessary with a typecast. This time, the channel is connected to the file *regions* – for the simple reason that the file size is small. The second channel is a usual output channel, which is connected to *System.out*, while the third is also an output channel connected to a file in the current directory.

The first statement in the *try* block creates an array with two *ByteBuffer* objects. This time I have used *allocateDirect()*, which is in principle the most effective buffer for files, but the important thing is that neither of the buffers can store the entire file *regions*. Note that the two buffers are not of the same size, what is not needed. The next statement performs a read on the input channel, wherein the array of buffers are a parameter. The *read()* method will read the input channel and fill the content into the two buffers.

The next *for* loop iterates to the two buffers, and for each iteration the buffer is fliped and drained to standard output, and finally is perform a *rewind()* so that the buffer can be drained again. After the loop the two buffers are drained to the output channel with the method *write()*, which here drains both buffers. That a channel in this way drains more buffers is called *gathering*. The output channel is created as another output channel, and it is therefore necessary with a typecast when the channel is created.

If the method is performed, you get results:

```
1081, Region Nordjylland
1082, Region Midtjylland
1083, Region Syddanmark
1084, Region Hovedstade
n
1085, Region Sj�lland
```

and you should notice how the first buffer is filled to its capacity, while the other only is partially filled. You may also notice that there is a small encoding error (*System.out*), but if you look at the file *regions1* in the current directory, you will see that the file is copied correctly.

FILE CHANNELS

FileChannel is a class that represents a channel to files. A *FileChannel* is an abstract class, but you can create an object by using the method *getChannel()*, which is supported by several of the classes in *java.io*. As an example is below shown a method that copies the file *municipalities* using *FileChannel* objects:

```
private static void test13()
{
  try (FileChannel src = (new FileInputStream(
    System.getProperty("user.home") + "/data/municipalities")).getChannel();
    FileChannel dst = (new FileOutputStream("municipalities3")).getChannel())
  {
    copy2(src, dst);
  }
  catch (IOException ex)
  {
    System.out.println("I/O error: " + ex.getMessage());
  }
}
```

The method is in principle the same as *test11()*, but the two *Channel* objects are created in a different way. The class *FileInputStream* have a method *getChannel()*, which returns a *FileChannel*. It is a read-only channel. Likewise, the class *FileOutputStream* has a method *getChannel()*, which returns a file channel that is a writeonly channel.

Similarly, the class *RandomAccessFile* has a method *getChannel()*, which returns a *FileChannel*, that is a readwrite channel. The class *FileChannel* has many methods, and I will mention the most important:

- *void force(boolean meta)* which enforces that all updates are written physically to the file. The parameter specifies where the file's metadata also should be written.
- *long position()* that returns the position of the physical file pointer.
- *FileChannel position(long newPosition)* that sets the position of the physical file pointer. With that method you can define where to read or write in the file.
- *int read(ByteBuffer buffer)* that reads bytes in the file and returns the number af bytes as read. The maximum number of bytes is determined by the capacity of the buffer.
- *int read(ByteBuffer buffer, long position)* which also reads bytes to buffer, but *postion* tells where in the file to start (the postion of the file pointer).
- *long size()* that returns the size of the file.
- *int write(ByteBuffer buffer)* that writes the content of the buffer to the file starting from the current position of the file pointer. The method returns the number og bytes written to the file.
- *int write(ByteBuffer buffer, long position)* which also writes bytes to the file, but *postion* tells where in the file to start (the postion of the file pointer).

As an example, is shown a method that writes to and reads from a *RandomAccessFile*:

```
public static void test14()
{
  try (FileChannel fc = (new RandomAccessFile("numbers", "rw")).getChannel())
  {
    System.out.println("Position = " + fc.position() + ", size = " + fc.size());
    ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
    buffer.putInt(23);
    buffer.putDouble(Math.PI);
    buffer.putInt(29);
    buffer.putDouble(Math.E);
    buffer.flip();
    fc.write(buffer);
    fc.force(false);
    System.out.println("Position = " + fc.position() + ", size = " + fc.size());
```

```
    buffer.clear();
    fc.position(0);
    fc.read(buffer);
    buffer.flip();
    System.out.println(buffer.getInt());
    System.out.println(buffer.getDouble());
    System.out.println(buffer.getInt());
    System.out.println(buffer.getDouble());
  }
  catch (Exception ex)
  {
    System.out.println("I/O error: " + ex.getMessage());
  }
}
```

First a *FileChannel* is opend to the file, and this time it's a readwrite channel, and the first thing that happens is that the method prints the position of the file pointer and file size. Next, a *ByteBuffer* is created. I now want to write an *int*, a *double*, an *int* and another *double* to this buffer. In order to format these values for bytes I exploit that a *ByteBuffer* has methods for this purpose, which partly copies the values into the buffer and moves the index *position* correctly. After the four values are written to the buffer a *flip()* is performed, and the buffer's contents are sent to the channel. Then empty the buffer and the file pointer is set 0, and the contents of the file are read to the buffer. Finally, the result is printed, and again are used the right get methods for reading from the buffer. If the method is performed, the result is:

```
Position = 0, size = 24
Position = 24, size = 24
23
3.141592653589793
29
2.718281828459045
```

MEMORY MAPPED FILES

A *FileChannel* has a method *map()* which makes it possible to embed an open file or a part of an open file in a buffer:

MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)

The three parameters denotes a mode in addition to the part of the file to be encapsulated (start position and the length of the segment to be encapsulated). Mode may be READ_ONLY or READ_WRITE. The type *MappedByteBuffer* has the same characteristics as other buffers, but its content is stored in a file. This means for example, that if you performs a *get()* and the file is updated by another program, so the changes are visible in the current program. The same applies to *put()* that if you update the buffer, the change will be visible in another programs that uses the same file. The goal of a memory mapped file is of course performance, and especially when processing large files like pictures. As an example is shown a method that creates a file with 1000 integers:

```
private static void test15()
{
  try (FileChannel fc = (new RandomAccessFile("numbers", "rw")).getChannel()
  {
    MappedByteBuffer buffer = fc.map(FileChannel.MapMode.READ_WRITE, 0, 4096);
    for (int i = 0; i < 1000; ++i) buffer.putInt(i + 1);
  }
```

```
  catch (IOException ex)
  {
    System.out.println("I/O error: " + ex.getMessage());
  }
}
```

The file is a *RandomAccessFile* and as in the previous example is created a *FileChannel* to the file. Next is created a *MemoryMappedBuffer* to the channel and the buffer says that the file will be mapped from start to byte 4096. This number corresponds exactly to that the file must contains 1000 integers of type *int*. The next loop fills the buffer with 1000 integers from 1 to 1000. You should note that the result of the buffer is written to the file that will contain the 1000 integers. Note also that since it is a difference row, the sum of the numbers are 500500.

The next method updates the file:

```
private static void test16()
{
  try (FileChannel fc = (new RandomAccessFile("numbers", "rw")).getChannel())
  {
    MappedByteBuffer buffer = fc.map(FileChannel.MapMode.READ_WRITE, 0, fc.size());
    for (int i = 0; i < 1000; ++i) buffer.putInt(2 * buffer.getInt(i * 4));
  }
  catch (IOException ex)
  {
    System.out.println("I/O error: " + ex.getMessage());
  }
}
```

Again is created a memory mapped file and all data in the file is updated by doubling. You should notice how the individual numbers are referenced:

```
buffer.getInt(i * 4)
```

returns an *int*, starting at the address *4*i*, but without moving the buffer's position. The statement

```
buffer.putInt(2 * buffer.getInt(i * 4));
```

then updates the same *int*, and then move the position 4 bytes forward.

Since all numbers are doubled, the sum of the numbers in the file then is 1001000. It is validated by the following method:

```
private static void test17()
{
  try (FileChannel fc = (new RandomAccessFile("numbers", "r")).getChannel())
  {
    long sum = 0;
    MappedByteBuffer buffer = fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());
    while (buffer.hasRemaining()) sum += buffer.getInt();
    System.out.println(sum);
  }
  catch (IOException ex)
  {
    System.out.println("I/O error: " + ex.getMessage());
  }
}
```

LOCKING FILES

As the last related to channels I will look at file locking. It is possible to put a lock on an entire file or just on a part of a file, where the lock indicates, where in the file locking begins of what area in the file to be locked. Locking occurs at process level (and not threads), and I will in this context distinguish between a *writer process* (a process that writes to the file) and a *reader process* (a process that simply reads from the file). There are two kinds of locks. A *writer process* can set an *exclusive lock* on a part of a file, and it can do so if there are no other processes that have put a lock that overlaps the part of the file to be locked. An exclusive lock gives the writer process full control over the locked part of the file such that it can both read and write. A reader process can put a *shared lock* on a part of the file, and it can do so if not another process has put an exclusive lock that overlaps the part of the file to be locked. This means that the several processes can set a shared lock for the same part of a file, and thus that more processes can read the file.

A lock has the type *FileLock* and the *FileChannel* class has four methods to create a lock:

1. *lock()*
2. *lock(long position, long size, boolean shared)*
3. *tryLock()*
4. *tryLock(long position, long size, boolean shared)*

The first and the third creates an exclusive lock on the entire file, while the other two can specify the part of the file to be locked and where the lock should be exclusive or shared. The difference is further that the first two blocks, which is to say that if they are unable to lock, they are set in a queue until they can get the lock, after which the process is running automatically again when the lock is released. The last two methods are not blocking, but if they can not create the lock, they will return *null*, and that is up to the program to decide what is to happen next.

The class *FileLock* has a number of methods that can return the information on the lock and its conditions, but most importantly is the method *release()* which is used to release the lock. Universally, one uses the following pattern to lock a file:

```
FileLock lock = fileChannel.lock();
try
{
  // interact with the file channel
}
catch (IOException ex)
{
```

```
  // handle the exception
}
finally
{
  lock.release();
}
```

and here it is important that the method *release()* is performed in a finally block, so that one is sure that it is executed.

To illustrate file locking in practice I have written following program *Locking*:

```
package locking;

import java.util.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class Locking
{
  private static final Random rand = new Random();
  private static final int N = 1000000;

  public static void main(String[] args)
  {
    boolean writer = args.length != 0;
    try (FileChannel fc =
      (new RandomAccessFile("integers", (writer) ? "rw" : "r").getChannel())
    {
      if (writer) write(fc); else read(fc);
    }
    catch (Exception ex)
    {
      System.out.println("I/O error: " + ex.getMessage());
    }
  }
  private static void read(FileChannel fc) throws Exception
  {
    for (int i = 0; i < N; i++)
    {
      System.out.println("Want a shared lock");
      long t1 = System.nanoTime();
      FileLock lock = fc.lock(0, 16, true);
      long t2 = System.nanoTime();
```

```
    System.out.println("Has a shared lock " + (t2 - t1));
    try
    {
      ByteBuffer buffer = ByteBuffer.allocate(16);
      fc.read(buffer, 0);
      buffer.flip();
      System.out.println("Sum = " +
       (buffer.getInt() + buffer.getInt() + buffer.getInt() + buffer.getInt()));
    }
    finally
    {
      lock.release();
      System.out.println("Shared lock released");
    }
  }
}
private static void write(FileChannel fc) throws Exception
{
  for (int i = 0; i < N; i++)
  {
    System.out.println("Want an exclusive lock");
    long t1 = System.nanoTime();
    FileLock lock = fc.lock();
    long t2 = System.nanoTime();
```

```
        System.out.println("Has an exclusive lock: " + (t2 - t1));
        try
        {
          ByteBuffer buffer = ByteBuffer.allocate(16);
          int value = rand.nextInt(89997) + 10000;
          System.out.println("Writing: " + value);
         buffer.putInt(value).putInt(value + 1).putInt(value + 2).putInt(value + 3);
          buffer.flip();
          fc.write(buffer, 0);
        }
        finally
        {
          lock.release();
          System.out.println("Exclusive lock released");
        }
      }
   }
}
```

The program is implemented as a command that may have an argument on the command line. The value does not matter, but in *main()*, a *boolean writer* is *true* or *false* depending on whether there is an argument or not. In *main()* is created a *RandomAccessFile* that is *readwrite* or *readonly* depending on the value of the variable *writer*, and the same variable is used to determine whether to perform a method *write()* or a method *read()*.

The method *write()* iterates a loop 1000000 times and for each iteration the method acquire an exclusive lock for the file. When the file has been locked the method prints, how long it has been waiting for the lock, and then writes 4 integers to file before releasing the lock again.

The method *read()* works in principle in the same way, just trying the method instead to get a shared lock, and instead of writing to the file it reads the 4 integers and prints their sum on the screen.

The idea is now to start two versions of the program, such they runs in each their terminal, and where the one writes, while the other reads. You will then be able to observe how the two programs alternately get a lock. It is important that the program that writes is started first.

3.3 PATH AND FILES

Paths represents the file system, and the class *Path* is playing the same role as the class *File* in *java.io*, but *Path* represents a more general concept and provides many more services available. In the following, a *Path* object refer a path to a file or a directory, but it does not have to be an existing file or directory, and so a *Path* is just a name.

Along with the class *Path* is also the class *Files*, and both classes are in the package *java.nio.file*. The class *Files* is very comprehensive, but generally it contains what is needed to manipulate the file system. The following are not be a complete documentation of these classes, and you are encouraged partly to examine the package *java.nio.file* and what types exist and specific the documentation for the class *Files*, so you have an idea of what this class makes available.

In the rest of this section and thus the rest of this chapter, I will through some simple testing methods show a little of what is possible and how to apply the two classes *Path* and *Files*.

THE CLASS PATH

A path is composed of a number of elements (file names) separated by a separator, which is a character that in Linux is /, while under Windows it is \. The following method prints information about a *Path* while also providing an introduction to some of the methods that the class *Path* makes available:

```
private static void print(Path p)
{
  System.out.println(p);                  // the path
  System.out.println(p.getFileName());    // the last element in the path
  System.out.println(p.getName(0));       // the 0th element in the path
  System.out.println(p.getNameCount());   // number of elements in the path
  System.out.println(p.subpath(0,2));     // the two first elements in the path
  System.out.println(p.getParent());      // the parent directory
  System.out.println(p.getRoot());        // the root directory
  System.out.println(p.toUri());          // the part converted to an URL
  try
  {
    Path path = p.toRealPath();
    System.out.println(path);
  }
  catch (Exception ex)
  {
    System.out.println(ex);
  }
  System.out.println(p.getFileSystem());
  System.out.print
ln("-------------------------------------------------------");
}
```

Below is a test method using the above method:

```
public static void test18()
{
  Path p1 = Paths.get("/home/pa/data");
  Path p2 = Paths.get("/home/pa/data/zipcodes");
  Path p3 = Paths.get("home", "pa", "data");
  Path p4 = Paths.get(FileSystems.getDefault().getSeparator(), "home", "pa",
    "data", "zipcodes");
  Path p5 = Paths.get(System.getProperty("user.home"));
  Path p6 = Paths.get("abc").toAbsolutePath();
  print(p1);
  print(p2);
  print(p3);
  print(p4);
  print(p5);
  print(p6);
}
```

The class *Paths* is a class that include a static method *get()*, which you can use to create a *Path* object. The first two statements are easy enough to understand and they creates (on my machine) a path to, respectively, a directory and a file. The next two statements does almost the same, but here you specify multiple arguments where the arguments are the elements of the path and the method then inserts the necessary separation characters. Actually it is recommend to use this syntax, as in this way makes the name regardless of the current operating system. The first, however, will create a relative path, while the other creates an absolute path, and the example should among other things show how to obtain the file system separation character and thus the symbol for root. The last two examples should show how to get a path to the home directory, and how to create an absolute path for a name. If the first path is printed, the result is:

```
/home/pa/data
data
home
3
home/pa
/home/pa
/
file:///home/pa/data/
/home/pa/data
sun.nio.fs.LinuxFileSystem@4e25154f
```

If you look at the results, there is not so much wonder in that, but you should note how there is created an URL for the name. The method performs the statement *toRealPath()*, which creates a *Path* object to an existing directory or file. This statment is placed in a try / catch, and if the file does not exist, you get an exeption. In this case, the file (the directory) exists and the path is printed. Finally, the last line is included only to show how to refer to the current file system.

The results corresponding to the variables *p2*, *p3*, *p4* and *p5* contains in principle nothing new, but for the variable *p3* is the result:

```
home/pa/data
data
home
3
home/pa
home/pa
null
file:///home/pa/doc/java/Java%205/source05/NIOProgram/home/pa/data
java.nio.file.NoSuchFileException: home/pa/data
sun.nio.fs.LinuxFileSystem@4e25154f
```

and here you must note that it is a relative file name, and that name's URL as the name of the current directory followed by the relative file name. You should also note that this time there is an exception when performing *toRealPath()* when the directory does not exist. Something similar is happened with *p6*:

```
/home/pa/doc/java/Java 5/source05/NIOProgram/abc
abc
home
8
home/pa
/home/pa/doc/java/Java 5/source05/NIOProgram
/
file:///home/pa/doc/java/Java%205/source05/NIOProgram/abc
java.nio.file.NoSuchFileException:
    /home/pa/doc/java/Java 5/source05/NIOProgram/abc
sun.nio.fs.LinuxFileSystem@4e25154f
```

abc is the name of a nonexistent file, and the method *toAbsolutPath()* creates a path which references to a file *abc* under current directory. The file is not existing, so you get back an exception when performing *toRealPath()*.

As another example the method below creates a *Path* object to a file named *numbers* and under my *tmp* directory. Next, the method uses the class *Files* that have a static method *newOutputStream()*, which creates an *OutputStream* to a file identified by a *Path* object. If the file does not exist, it is created (if possible), and then there is written 100 bytes to the file:

```
private static void test19()
{
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/numbers");
  try (OutputStream out = new BufferedOutputStream(Files.newOutputStream(path)))
  {
    for (byte b = 0; b < 100; ++b) out.write(b);
  } catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

THE CLASS FILES

The class *Files* is as mentioned, very comprehensive and has among other things a number of static methods that can print information about a files attributes:

```
private static void test20()
{
  info(Paths.get("/home/pa/tmp"));
  info(Paths.get("/home/pa/tmp/numbers"));
}

private static void info(Path path)
{
  System.out.println(Files.exists(path));
  System.out.println(Files.isDirectory(path));
  System.out.println(Files.isRegularFile(path));
  System.out.println(Files.isReadable(path));
  System.out.println(Files.isWritable(path));
  System.out.println(Files.isExecutable(path));
  System.out.println(Files.isSymbolicLink(path));
  System.out.print
ln("--------------------------------------------------");
}
```

where *numbers* are the file created above. The meaning of each method should follow from the name, and is the method perform is the result:

```
true
true
false
true
true
true
false
-----------------------------------------------------
true
false
true
true
true
false
false
-----------------------------------------------------
```

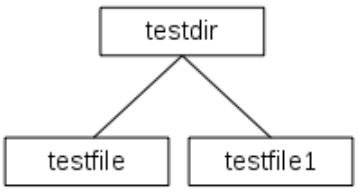
Consider then the following method:

```
private static void test21()
{
  try
  {
    Path p0 = Paths.get(System.getProperty("user.home"));
    Path p1 = Files.createDirectory(p0.resolve("testdir"));
    Path p2 = Files.createFile(p1.resolve("testfile"));
    System.out.println(p0);
    System.out.println(p1);
    System.out.println(p2);
    Files.copy(p2, p1.resolve("testfile1"));
    System.out.println(Files.deleteIfExists(p1.resolve("testfile1")));
    System.out.println(Files.deleteIfExists(p2));
    System.out.println(Files.deleteIfExists(p1));
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

The method creates a path *p0* to my home directory. With this path available means

```
p0.resolve("testdir")
```

the creation of a path where the name *testdir* becomes a "file" under *p0* – in this case a subdirectory. *Files* has a static method *createDirrectory()*, which creates a directory from a path and returns a *Path* object *p1* to this directory. Likewise, *p2* is a path to a file in this directory. *Files* also has a method *copy()* to copy a file. In this case, the file that is copied has the path *p2* and is copied to a file in the same directory, but with the name *testfile1*. The result of performing these statements is that in my home directory is created following file tree:



The last three statements of the method deletes this file tree again, and is the method executed, the result is:

```
/home/pa
/home/pa/testdir
/home/pa/testdir/testfile
true
true
true
```

The following method shows how to create a path to a file in my *tmp* directory, and then open an *OutputStream* with this path and prints 15 bytes to the stream:

```
private static void test22()
{
  byte[] b = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 };
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file1");
  try (OutputStream out = Files.newOutputStream(path, StandardOpenOption.CREATE))
  {
    out.write(b);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

There is not much to explain, but you should notice how you can set an option to *newOutputStream()*. *StandardOpenOption* is an enumeration, and you are encouraged to study the other options.

This same problem can also be solved as follows:

```
private static void test23()
{
  byte[] b = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 };
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file2");
  try
  {
    Files.write(path, b, StandardOpenOption.CREATE);
  }
  catch (Exception ex)
  {
    System.out.println(ex);
  }
}
```

where the method *write()* opens the file, writes a byte array to the file and close it again. Below are two methods that read the two files again:

```
private static void test24()
{
  byte[] b = new byte[100];
```

```
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file1");
  try (InputStream in = Files.newInputStream(path))
  {
    int count = in.read(b);
    for (int i = 0; i < count; ++i) System.out.print(b[i] + " ");
    System.out.println();
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}

private static void test25()
{
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file2");
  try
  {
    byte[] b = Files.readAllBytes(path);
    for (int i = 0; i < b.length; ++i) System.out.print(b[i] + " ");
    System.out.println();
  }
  catch (Exception ex)
  {
    System.out.println(ex);
  }
}
```

The next method shows how you with a single statement can write a list of strings to a file:

```
private static void test26()
{
  List<String> lines = Arrays.asList("Gorm den Gamle", "Harald Blåtand",
      "Svend Tveskæg", "Harald d. 2.", "Knud den Store", "Hardeknud",
      "Magnus den Gode");
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file3");
  try
  {
   Files.write(path, lines, Charset.defaultCharset(), StandardOpenOption.CREATE);
  }
  catch (Exception ex)
  {
    System.out.println(ex);
  }
}
```

The method *write()*, this time has four parameters:

```
write(Path path, Iterable<? extends CharSequence> lines, Charset cs,
    OpenOption … options)
```

where the first is the path, the next a collection of objects derived from *CharSequence* (and, in practice it is *String*), the third a character encoding and finally the possibility of multiple options. It is very simple to create a text file.

Below is a method that reads the file again:

```
private static void test27()
{
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file3");
  try
  {
    List<String> list = Files.readAllLines(path, Charset.defaultCharset());
    for (String line : list) System.out.println(line);
  }
  catch (Exception ex)
  {
    System.out.println(ex);
  }
}
```

```
Gorm den Gamle
Harald Blåtand
Svend Tveskæg
Harald d. 2.
Knud den Store
Hardeknud
Magnus den Gode
```

The following methods shows the same, but instead they uses respectively a *BufferedWriter* and a *BufferedReader*, and also are shown how you can specify the encoding:

```
private static void test28()
{
  List<String> lines = Arrays.asList("Gorm den Gamle", "Harald Blåtand",
    "Svend Tveskæg", "Harald d. 2.", "Knud den Store", "Hardeknud",
    "Magnus den Gode");
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file4");
  try (BufferedWriter writer =
      Files.newBufferedWriter(path, Charset.availableCharsets().get("UTF-8")))
  {
```

```
    for (String line : lines)
    {
      writer.write(line);
      writer.newLine();
    }
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}

private static void test29()
{
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file4");
  try (BufferedReader reader =
     Files.newBufferedReader(path, Charset.availableCharsets().get("UTF-8")))
  {
    for (String line = reader.readLine(); line != null; line = reader.readLine())
      System.out.println(line);
  }
  catch (IOException ex)
  {
```

```
      System.out.println(ex);
    }
}
```

The most important of these examples is how you with the class *Files* can create both a *BufferedWriter* and a *BufferedReader*.

The next two examples are similar and shows how the class *Files* can create a *BufferedOutputStream* and a *BufferedInputStream*:

```
private static void test30()
{
  byte data[] = new byte[100];
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file5");
  try (OutputStream out = new BufferedOutputStream(Files.newOutputStream(
    path, StandardOpenOption.CREATE, StandardOpenOption.APPEND)))
  {
    rand.nextBytes(data);
    for (int i = 0; i < 100; ++i) out.write(data);
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}

private static void test31()
{
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file5");
  long sum = 0;
  try (InputStream in = new BufferedInputStream(Files.newInputStream(path)))
  {
    for (int t = in.read(); t != -1; t = in.read()) sum += t;
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
  System.out.println(sum);
}
```

The first method writes 10000 bytes to a *BufferedOutputStream*, and here you should especially note that it is opened in append mode. The method writes 100 times, and each time it writes an array with 100 random bytes. The last method reads the file again by means of a *BufferedInputStream* and determines the sum.

As the last example I will show how to write strings to a file, but this time by using a channel:

```
private static void test32()
{
  List<String> lines = Arrays.asList("Gorm den Gamle", "Harald Blåtand",
    "Svend Tveskæg", "Harald d. 2.", "Knud den Store", "Hardeknud",
    "Magnus den Gode");
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file6");
  try (ByteChannel channel = Files.newByteChannel(path,
    StandardOpenOption.CREATE, StandardOpenOption.WRITE))
  {
    for (String line : lines)
    {
      ByteBuffer buffer =
        ByteBuffer.wrap((line + '\n').getBytes(Charset.forName("UTF-8")));
      channel.write(buffer);
    }
  }
  catch (IOException ex)
  {
    System.out.println(ex);
  }
}
```

First is created a list with the names of the same 7 Danish kings as used above and then a path to a file in my *tmp* directory is created. The class *Files* has a method *newByteChannel()*, which creates a *ByteChannel* for a file identified by a path. With this channel ready the method performs a loop that iterates over all strings in the list. For each string (name) are added a line break, and all are converted to a byte array after an UTF-8 encoding. This array is finally encapsulated in a *ByteBuffer* that is send out on the channel.

Below is the equivalent method, which reads the file again:

```
private static void test33()
{
  Path path = Paths.get(System.getProperty("user.home")).resolve("tmp/file6");
  try (ByteChannel channel = Files.newByteChannel(path, StandardOpenOption.READ))
  {
    ByteBuffer buffer = ByteBuffer.allocate(30);
    while (channel.read(buffer) > 0)
    {
      buffer.flip();
      System.out.print(Charset.forName("UTF-8").decode(buffer));
      buffer.clear();
    }
```

```
      System.out.println();
    }
    catch (IOException ex)
    {
      System.out.println(ex);
    }
}
```

There is again created a *ByteChannel* to the file, and then a *ByteBuffer*. The number 30 is arbitrary, but is, however, selected so that the buffer is not large enough to contain the longest name. More precisely, this means that the subsequent read on the channel reads the 30 bytes at a time until there is fewer than 30 bytes. Every time the buffer is full, or more exactly when the *read()* method is performed the program executes a *flip()* and the byte content of the buffer is decoded to a *String*, which is printed on the screen.

4 OPERATIONS ON SIMPLE DATA TYPES

Any data element (simple variable or object) uses space in the machine's memory, and when the smallest addressable unit in the machine's memory is a byte, uses all data elements without exception a whole number of bytes. What it is depends on the element and is determined by the element's data type. A byte consists of 8 bits (that is 8 of the symbols 0 and 1), and thus you can consider a byte as a bit pattern consisting of 8 bits. Since a bit can take on two values, a byte thus can represent different values. How these values are interpreted depends entirely on the program and the data elements data types, and in some contexts it will be as a number, while in other contexts would be a code for a character.

In this chapter I will in detail describe how the simple data types are represented in Java as bit patterns and in this context also what operations are available to manipulate the individual bits. The chapter assumes knowledge of the binary numbers, and also the hexadecimal numbers, and so far you lack the knowledge referring to the book's appendix.

4.1 THE INTEGERS

Java has as already described four types of integers: *byte*, *short*, *int* and *long*, and they differ only with respect to what they fill in memory, which are respectively 1, 2, 4 or 8 bytes. As an example an *int* uses 4 bytes, and is in memory allocated as 4 contiguous bytes. This means that an *int* uses 32 bits and can therefore represent 4294967296 different integers as there with 32 bits is 4294967296 different bit patterns (2^{32}). The 32 bits are interpreted as a binary number, and where negative numbers are represented by their 2 complement. This corresponds to, that an *int* represents the integers in the range [a, b] where

```
a = -2147483648 (the binary number 10000000000000000000000000000000)
b = 2147483647 ( the binary number 01111111111111111111111111111111)
```

Any integer type supports the binary operations:

- | bitvis or
- & bitvis and
- ^ bitvis xor
- ~ 1 complement or just complement
- << left shift
- >> arithmetic right shift
- >>> right shift

The representation of integers is simple, and it is easy test the above in Java. The program *BinProgram* contains the following method:

```
public static String toBin(int t)
{
  StringBuilder builder = new StringBuilder();
  for (int i = 0; i < 32; ++i, t <<= 1)
    builder.append((t & 0x80000000) == 0 ? '0' : '1');
  return builder.toString();
}
```

that converts an integer to a 32-bit string. Note that the class *Integer* has a method *toBinaryString()*, which returns an *int* as a bit string, but the method does not preceds by 0s, so the above method. If you perform the following method

```
private static void test01()
{
  print(113);
  print(-113);
  print(0);
  print(-1);
  print(Integer.MAX_VALUE);
  print(Integer.MIN_VALUE);
}

private static void print(int t)
{
  System.out.println(toBin(t) + " = " + t);
}
```

you get the result

```
00000000000000000000000001110001 = 113
11111111111111111111111110001111 = -113
00000000000000000000000000000000 = 0
11111111111111111111111111111111 = -1
01111111111111111111111111111111 = 2147483647
10000000000000000000000000000000 = -2147483648
```

corresponding to what is said above. Likewise the method:

```
private static void test02()
{
  int a = 0xF0F0F0F0;
  int b = 0x3C3C3C3C;
  System.out.println(toBin(a));
```

```
    System.out.println(toBin(b));
    System.out.println(toBin(a | b));
    System.out.println(toBin(a & b));
    System.out.println(toBin(a ^ b));
    System.out.println(toBin(~a));
    System.out.println(toBin(a << 3));
    System.out.println(toBin(a >> 3));
    System.out.println(toBin(a >>> 3));
}
```

results in the following:

```
11110000111100001111000011110000
00111100001111000011110000111100
11111100111111001111110011111100
00110000001100000001100000001100
11001100110011001100110011001100
00001111000011110000111100001111
10000111100001111000011110000000
11111110000111100001111000011110
00011110000111100001111000011110
```

Here you should specifically note that the two variables *a* and *b* are initialized using hexadecimal values that are useful when working with binary integers. Note also that the operator >> is an arithmetic right shift (is the sign bit which is added), but the >>> operator is a right shift, that always inserts a 0.

As for the other types to integers it is only the ranges, that the types represents that are different, and therefore how big numbers to work with. However, the binary operations sometimes give a different result than expected. If an argument is a *byte* or *short*, the argument will be converted to an *int* before the operation is performed and the result will be a 32 bit *int*. If the argument represents a negative number it may result in a different outcome than expected. Something similar may happen if an argument is a *long* as the second argument then must be converted to a *long* before the operation is performed. Consider as an example the following method:

```
private static void test03()
{
  byte b = -56;
  byte b1 = (byte)(b >> 3);
  byte b2 = (byte)(b >>> 3);
  byte b3 = (byte)((b & 0xff) >>> 3);
  System.out.println(toBin(b));
  System.out.println(toBin(b1));
  System.out.println(toBin(b2));
  System.out.println(toBin(b3));
}
```

If the method is performed, you get the result:

```
11111111111111111111111111001000
11111111111111111111111111111001
11111111111111111111111111111001
00000000000000000000000000011001
```

The value of *b* is 11001000, and when it is transferred to *tobin()*, it will be expanded to 32 bits and are filled with the sign. The first result is as expected. This also applies to the second result when it is an arithmetic shift, but you should note that the operation is performed by first expanding *b* to 32 bits and then performs an arithmetic shift. The third result is however not what one would expect, since it is a non arithmetic shift, but the following occur:

b is expanded to 32 bits: 11111111111111111111111111001000
a right shift on 3 bits: 00011111111111111111111111111001
a type cast to a *byte*: 11111001
expanded to 32 bits with a call to *toBin()*: 11111111111111111111111111111001

The problem can be solved by expanded *b* to 32 bits:

```
b && 0xff =
11111111111111111111111111001000 & 00000000000000000000000011111111 =
00000000000000000000000011001000
```

EXERCISE 6

Add a class named *Binary* to the class library *PaLib* when the class must be part of the package *palib.util*. The class must have the following methods:

```
package palib.util;

public class Binary
{
  /**
   * Converts the argument to a binary string of 8 bits.
   * @param b The byte to be converted
   * @return The argument converted to a binary string of 8 bits
   */
  public static String toBin(byte b) { … }

  /**
   * Converts the argument to a binary string of 16 bits.
   * @param t The integer to be converted
   * @return The argument converted to a binary string of 16 bits
   */
  public static String toBin(short t) { … }

  /**
   * Converts the argument to a binary string of 32 bits.
   * @param t The integer to be converted
   * @return The argument converted to a binary string of 32 bits
   */
  public static String toBin(int t) { … }
```

```
/**
 * Converts the argument to a binary string of 64 bits.
 * @param t The integer to be converted
 * @return The argument converted to a binary string of 64 bits
 */
public static String toBin(long t) { … }

/**
 * Converts the argument to a binary string of 16 bits.
 * @param t The integer to be converted
 * @param c Seperation character between byte values
 * @return The argument converted to a binary string of 16 bits
 */
public static String toBin(short t, char c) { … }

/**
 * Converts the argument to a binary string of 32 bits.
 * @param t The integer to be converted
 * @param c Seperation character between byte values
 * @return The argument converted to a binary string of 32 bits
 */
public static String toBin(int t, char c) { … }
```

```
  /**
   * Converts the argument to a binary string of 64 bits.
   * @param t The integer to be converted
   * @param c Seperation character between byte values
   * @return The argument converted to a binary string of 64 bits
   */
  public static String toBin(long t, char c) { … }
}
```

Also, write a program that can test the methods in the new class.

Java supports as mentioned above, the binary operations, including in particular the left and right shift. As other binary operations are rotations. If, for example

b = 11001000

is

```
b = 00110010
```

a binary right rotation on 2, while

```
b = 00100011
```

is a binary left rotation on 2. You must now expand the class *Binary* with the following methods:

```
/**
 * Rotates the argument b n bits to the right.
 * @param b The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument b rotated n bits to the right
 */
public static byte ror(byte b, int n) { … }

/**
 * Rotates the argument t n bits to the right.
 * @param t The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument t rotated n bits to the right
 */
public static short ror(short t, int n) { … }
```

```
/**
 * Rotates the argument t n bits to the right.
 * @param t The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument t rotated n bits to the right
 */
public static int ror(int t, int n) { … }

/**
 * Rotates the argument t n bits to the right.
 * @param t The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument t rotated n bits to the right
 */
public static long ror(long t, int n) { … }

/**
 * Rotates the argument b n bits to the left.
 * @param b The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument b rotated n bits to the left
 */
public static byte rol(byte b, int n) { … }

/**
 * Rotates the argument b n bits to the left.
 * @param t The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument b rotated n bits to the left
 */
public static short rol(short t, int n) { … }

/**
 * Rotates the argument b n bits to the left.
 * @param t The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument b rotated n bits to the left
 */
public static int rol(int t, int n) { … }

/**
 * Rotates the argument b n bits to the left.
 * @param t The argument to be rotated
 * @param n The number of bits to be rotated
 * @return The argument b rotated n bits to the left
 */
public static long rol(long t, int n) { … }
```

You also you need to expand the test program to test the methods for binary rotation.

EXERCISE 7

Java defines a class called *BitSet* and which, in principle, represents a set, but as a so-called bitmap, which can be thought of as an array consisting of elements of the type bits, and thus elements that are 0, or 1. You must in this exercise write a program named *BitmapProgram* that has three test methods for the class *BitSet*.

The first method must create a *BitSet* and tests the methods *size()*, *cardinality()*, *set()* and *toString()*.

Add a method

```
static String toBin(BitSet s)
{
}
```

that returns a bit string for the *BitSet s*.

You must then write a second test method, when the method must create a *BitSet* with the values 2, 3, 5, 7, 11, 13, 17, 19, 23 and 29, and then the method must print the corresponding bit string by means of the above *toBin( )* method.

Write finally a test method that creates two *BitSet* objects initialized with appropriate values. The test method should then try the *BitSet* class's *and()*, *notAnd(), or()* and *xor()* methods.

PROBLEM 3

A BitSet is an example of a bitmap, and in this problem you should write a class *Bitmap* that represents a bitmap and is an alternative to the Java's *BitSet*. A bitmap is, in principle, a sequence of bits:

```
0010111010101010101011110101010100111111101010101011111000100001
```

where there is 64 bits. Such a bitmap can, for example be implemented as eight *byte*s, 2 *int*s or a *long*. A bitmap must, however, be more general and should be able to be of any size, and therefore it should be implemented as an array of one of the simple types to integers. Immediately it is simple to implement a bitmap, but there are several things you must consider and make a decision. As important examples I will mention:

- which type to be used to the internal array for the individual bits
- must the bitmap have fixed size
- which sizes should be possible for a bitmap and should it be a multiplum of the size of the array's elements
- which operations must a bitmap make available
- when are two bitmaps compatible

As regards of the first, you can choose between *byte*, *short*, *int* and *long*. In the case of a large bitmap, you should select a type with many bits as it means a smaller array, and thus shorter loops in the methods, providing increased efficiency. In the case of small bitmaps, a 64 bits element can means unnecessary space consumption, but in most practical cases where you need a bitmap, it will be bitmaps with many bits, and it indicates to choose a type with room for many bits. Below you should select the type *long*.

As for the size, it is the application that determines what is the best choice. In this task, the size must be fixed, where you have to specify the size as a parameter to the constructor. It must be able to have any logical size – for example 70 bits. If it is implemented as an array of the type *long* it will occupies 128 bits, but there should only be access (reference) to the first 70.

Based on the above, it is your task to expand the class library *PaLib* with the following class:

```
package palib.util;

import java.util.*;

/**
 * Implements a bitmap consisting of an arbitrary number of bits.
 * The number of bits (size) is fixed and defined when creating an object.
 * Two bitmaps are compatible only if they consist of the same number of bits.
 */
public class Bitmap implements Comparable<Bitmap>

{
  /**
   * Creates a bitmap of size bits. All bits is 0.
   * @param size The size of the bitmap
   */
  public Bitmap(int size) { … }

  /**
   * Creates a bitmap initialized with a BitSet. The size is determined by the
   * logic value of s, and the individual bits are initialized corresponding to
   * the elements of s.
   * @param s The BitSet, to initialize this bitmap
   */
  public Bitmap(BitSet s) { … }

  /**
   * Creates a bitmap, which is initialized with a byte array.
   * @param arr The array used to initialize this bitmap
   */
  public Bitmap(byte[] arr) { … }

  /**
   * Copy constructor.
   * @param bm The bitmap, that must be copied
   */
  public Bitmap(Bitmap bm) { … }

  /**
   * Returns the size of this bitmap
   * @return The size of this bitmap
   */
  public int getSize() { … }
```

```
/**
 * Returns where the n'th bit is 1.
 * @param n Index of the bit to be tested
 * @return true, if the n'th bit is 1
 */
public boolean get(int n) { … }

/**
 * Sets the n'th bit in the bitmap to 1 or 0. Sets the n'th bit to 1 if the value
 * is true and 0 otherwise.
 * @param n Index of the bit to be set
 * @param value true means 1 and false 0
 */
public void set(int n, boolean value) { … }

/**
 * Sets all bits to 0.
 */
public void clear() { … }
```

```
 /**
  * Swap all bits in this bitmap.
  * @return A reference to the current bitmap
  */
public Bitmap flip() { … }

/**
  * Performs an AND between the bitmap and the parameter.
  * The result is that the state of this bitmap is changed.
  * @param bm The bitmap that this bitmap are AND'ed with
  * @return A reference to the current bitmap
  * @throws UtilException If the two bitmaps are different lengths
  */
public Bitmap and(Bitmap bm) throws UtilException { … }

/**
  * Performs an OR between the bitmap and the parameter.
  * The result is that the state of this bitmap is changed.
  * @param bm The bitmap that this bitmap are OR'ed with
  * @return A reference to the current bitmap
  * @throws UtilException If the two bitmaps are different lengths
  */
public Bitmap or(Bitmap bm) throws UtilException { … }

/**
  * Performs an XOR between the bitmap and the parameter.
  * The result is that the state of this bitmap is changed.
  * @param bm The bitmap that this bitmap are XOR'ed with
  * @return A reference to the current bitmap
  * @throws UtilException If the two bitmaps are different lengths
  */
public Bitmap xor(Bitmap bm) throws UtilException { … }

/**
  * Performs a left shift on the current bitmap on n bits
  * @param n The number of bits to shift
  * @return A reference to the current bitmap
  */
public Bitmap shl(int n) { … }

/**
  * Performs a right shift on the current bitmap on n bits.
  * It is a non-arithmetic shift.
  * @param n The number of bits to shift
  * @return A reference to the current bitmap
  */
public Bitmap shr(int n) { … }
```

```
/**
 * Performing a comparison of the current bitmap by parameter bm. The first bit,
 * where the two bitmaps are different, determines the order, so that 1
 * is considered high.
 * @param bm The bitmap to be compared with
 * @return 1 if the current bitmap is the largest, -1 if it is less than,
 * and 0 if there are the same
 */
@Override
public int compareTo(Bitmap bm) { … }

/**
 * Overriding toString() so that it returns a string of the bits in the bitmap
 * @return A bit string representing the bits in the bitmap
 */
@Override
public String toString() { … }

/**
 * Overloading of equals().
 * @param obj The object to be compared with
 * @return true, if the current bitmap is the same value as the obj interpreted
 * as a bitmap
 */
@Override
public boolean equals(Object obj) { … }

/**
 * Overriding hash code, which is done by performers an xor of this bitmaps 32
 * bit words.
 * @return A hash code for this bitmap
 */
@Override
public int hashCode() { … }

/**
 * Returns the value of this bitmap as a BitSet.
 * @return The value of this bitmap as a BitSet.
 */
public BitSet toBitSet() { … }
/**

 * Returns this bitmap as a byte array. If the length of the bitmap do not
 * specify an exact number of bytes the last byte id filled with 0 bits.
 * @return This bitmap as a byte array
 */
public byte[] toBytes() { … }
```

```
/**
 * Static method returns a bitmap, which is a copy of the parameter bm, but with
 * all the bits inverted.
 * @param bm The bitmap to be returned a flip of
 * @return A flip of the parameter bm
 */
public static Bitmap flip(Bitmap bm) { … }

/**
 * Static method returns a bitmap, that is an AND of the bm1 and bm2.
 * @param bm1 The first argument to the binary operation
 * @param bm2 The second argument to the binary operation
 * @return An and of bm1 and bm2
 * @throws UtilException If the two bitmaps are of different length
 */
public static Bitmap and(Bitmap bm1, Bitmap bm2) throws UtilException { … }
```

```
/**
 * Static method returns a bitmap, that is an OR of the bm1 and bm2.
 * @param bm1 The first argument to the binary operation
 * @param bm2 The second argument to the binary operation
 * @return An or of bm1 and bm2
 * @throws UtilException If the two bitmaps are of different length
 */
public static Bitmap or(Bitmap bm1, Bitmap bm2) throws UtilException { … }

/**
 * Static method returns a bitmap, that is a XOR of the bm1 and bm2.
 * @param bm1 The first argument to the binary operation
 * @param bm2 The second argument to the binary operation
 * @return A xor of bm1 and bm2
 * @throws UtilException If the two bitmaps are of different length
 */
public static Bitmap xor(Bitmap bm1, Bitmap bm2) throws UtilException { … }

/**
 * Static method returns a bitmap, which is a copy of the parameter bm shifted n
 * places to the left.
 * @param bm The bitmap that is copied and changed
 * @param n The number of bits to be shifted
 * @return A copy of bm shifted n places to the left
 */
public static Bitmap shl(Bitmap bm, int n) { … }

/**
 * Static method returns a bitmap, which is a copy of the parameter bm shifted n
 * places to the right. It is a non-arithmetic shift.
 * @param bm The bitmap that is copied and changed
 * @param n The number of bits to be shifted
 * @return A copy of bm shifted n places to the right
 */
public static Bitmap shr(Bitmap bm, int n) { … }
```

You should also expand the class *Binary* with a new method:

```
/**
 * Converts a bitmap to a binary string where the individual bytes are separated
 * by the character c.
 * @param bm The bitmap to be converted
 * @param c Seperation character between byte values
 * @return bm converted to a bit string
```

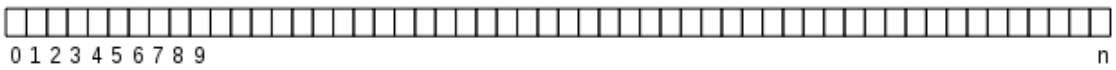
```
 */
public static String toBin(Bitmap bm, char c)
{
}
```

When you have written the class (and the above methods), you must write a test program (for example *TestBitmap*) that can test the class.

PROBLEM 4

As an example of using a bitmap, you must write a program that uses Eratosthenes's algorithm to determine the primes. The idea is the following:

In order to determine all prime numbers that are less than or equal to N, start by defining a bitmap with N+1 elements:



In the beginning all positions are 0. You then starts with putting a mark (a 1 bit) in position 0 and position 1 as well as all the positions where the index is divisible by 2 – though not position 2:

In this run has been set a mark in all the even numbers greather than 2, as they are not primes. You then repeats it all, but this time you put a mark on the positions where the index is divisible by 3 – though not position 3:

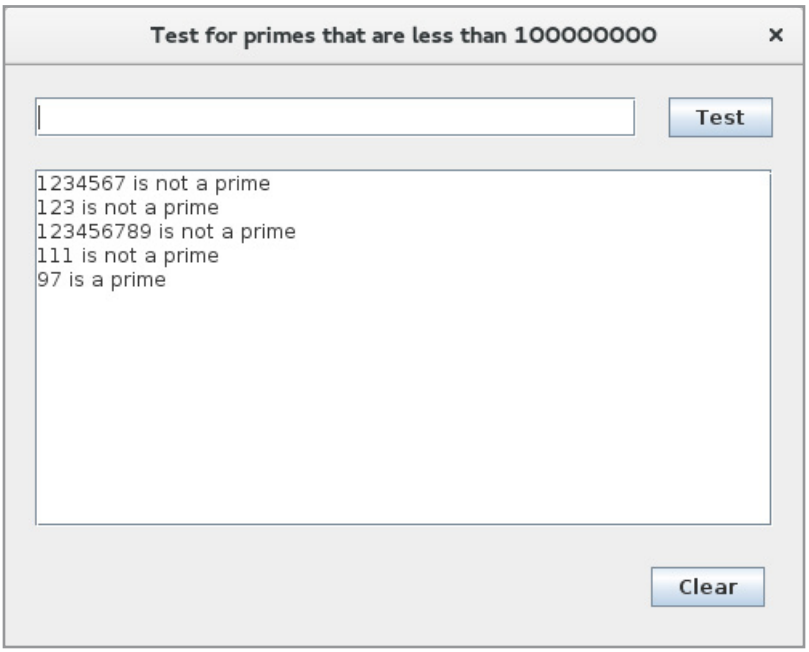
All positions where the index has 3 as the prime factor is now marked. Next time you repeat it all with the number 5 (the first position after 3 that is 0). All positions where the index has 5 as the prime factor is marked:

To resume. Next time start with the first position, which is not marked (it is position 7) and it is a prime, since it would otherwise be marked by a previous cycle. When you have been through it all, all positions with an index that is not prime are checked.

You must now write a program where the user can enter a number (se the window below). When you click the *Test* button, the application must insert a line in the list box that shows whether the number is a prime number. To determine if this is the case, the program must use the above algorithm and the class *Bitmap* from above. You can improve the algorithm a bit by observing the following:

1. If a number has a prime factor (a prime number that divides the numbert), it or another prime factor must be less than or equal to the square root of the number, and the above iteration can therefore stop when starting index is greater than square root of the number.
2. If you have reached the indexthat is not marked in a previous cycle, is$k$is a prime, and you have to mark all the numbers that has$k$as divisor: $2k$, $3k$, $4k$, …, then it actually is enough to start withk^2, because the position else would be marked by a previous cycle.
3. It is not necessary in the bitmap to track other than the odd numbers, because all the even numbers greater than 2 are not primes.

Eratosthenes algorithm is actually a very effective prime method, but the problem is of course that the bitmap take place in memory – even if you use the above observations.



EXERCISE 8

Create a project that you can call *Encoding*. Add the following two test methods, where the first write a text to a file, while the other reads the text again:

```
private static void test1()
{
  try
  {
    BufferedWriter writer = new BufferedWriter(new FileWriter("test1"));
    writer.write("Søren Sørensen\nPistolStræde 19\n6666 Borremose");
    writer.close();
  }
  catch (Exception ex)
  {
  }
}

private static void test2()
{
  try
  {
    BufferedReader reader = new BufferedReader(new FileReader("test1"));
    for (String line = reader.readLine(); line != null; line = reader.readLine()
      System.out.println(line);
```

```
    reader.close();
  }
  catch (Exception ex)
  {
  }
}
```

Test the two methods from the *main()* method. Do they have the expected result?

Write two other test methods:

```
private static void test3()
{
  try
  {
    BufferedWriter writer = new BufferedWriter(
      new OutputStreamWriter(new FileOutputStream("test2"), "UTF-8"));
    writer.write("Søren Sørensen\nPistolStræde 19\n6666 Borremose");
    writer.close();
  }
  catch (Exception ex)
  {
  }
}

private static void test4()
{
  try
  {
    BufferedReader reader = new BufferedReader(new InputStreamReader(
      new FileInputStream("test2"), "UTF-8"));
   for (String line = reader.readLine(); line != null; line = reader.readLine())
      System.out.println(line);
    reader.close();
  }
  catch (Exception ex)
  {
  }
}
```

Note that the difference is that this time the file is opened in a different manner (with an additional parameter), and that the file has another name. Test also these two methods from *main()* and note that you get the expected result.

Write two more test methods (which you can call *test5()* and *test6()*), where the differences only is that the file has a different name (this time *test3*) and a second parameter concerning encoding:

```
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
  new FileOutputStream("test3"), "ISO-8859-1"));
BufferedReader reader = new BufferedReader(new InputStreamReader(
  new FileInputStream("test3"), "ISO-8859-1"));
```

Test also these two methods. Write finally two test methods test7() and *test8()*, where the files are opened as follows:

```
BufferedWriter writer = new BufferedWriter(
  new OutputStreamWriter(new FileOutputStream("test4"), "UTF-16"));
BufferedReader reader = new BufferedReader(
  new InputStreamReader(new FileInputStream("test4"), "UTF-16"));
```

and try also these two methods. If you uses *Files* to examines the four files, you can see how much they fills on the disk:

- *test1* 48 bytes
- *test2* 48 bytes
- *test3* 45 bytes
- *test4* 92 bytes

Can you explains this 4 numbers?

EXERCISE 9

Create a project that you can call *FloatingPoint*. The two wrapper classes *Float* and *Double* have two constants *MIN_VALUE* and *MAX_VALUE*. Print these four constants on the screen and check how it fits with what has been said in the appendix (they're actually not quite).

You must then open the project for *PaLib* and add two methods to the class *Binary*:

```
/**
 * Converts the argument to a binary string of 32 bits.
 * @param t The number to be converted
 * @return The argument converted to a binary string of 32 bits
 */
public static String toBin(float t)
{
}

/**
 * Converts the argument to a binary string of 64 bits.
 * @param t The number to be converted
 * @return The argument converted to a binary string of 64 bits
 */
public static String toBin(double t)
{
}
```

In fact, it is trivial because the two wrapper classes has the methods *Float.floatToRawIntBits()* and *Double.doubleToRawLongBits()* that you can use. Write equivalent methods, where you can insert a separation character between the individual bytes:

```
public static String toBin(float t, char c)
{
}
```

```
public static String toBin(double t, char c)
{
}
```

Use these methods to print the binary representation of the numbers 123.45F, -123.45F, 12345.6789 and -12,345.6789 and check if it fits to what is said in the appendix.

The class *Float* has three further constants: *NEGATIVE_INFINITY*, *POSITIVE_INFINITY* and *NaN*. Print the binary representation of these constants. Do the same for the class *Double*.

EXERCISE 10

In problem 3 you you have written a class *Bitmap*, which is located in your class library *PaLib*. The class is included in the package *palib.util*, and to this package you must add the following exception class:

```
package palib.util;
public class UtilException extends Exception
{
  public UtilException(String message)
  {
    super(message);
  }
}
```

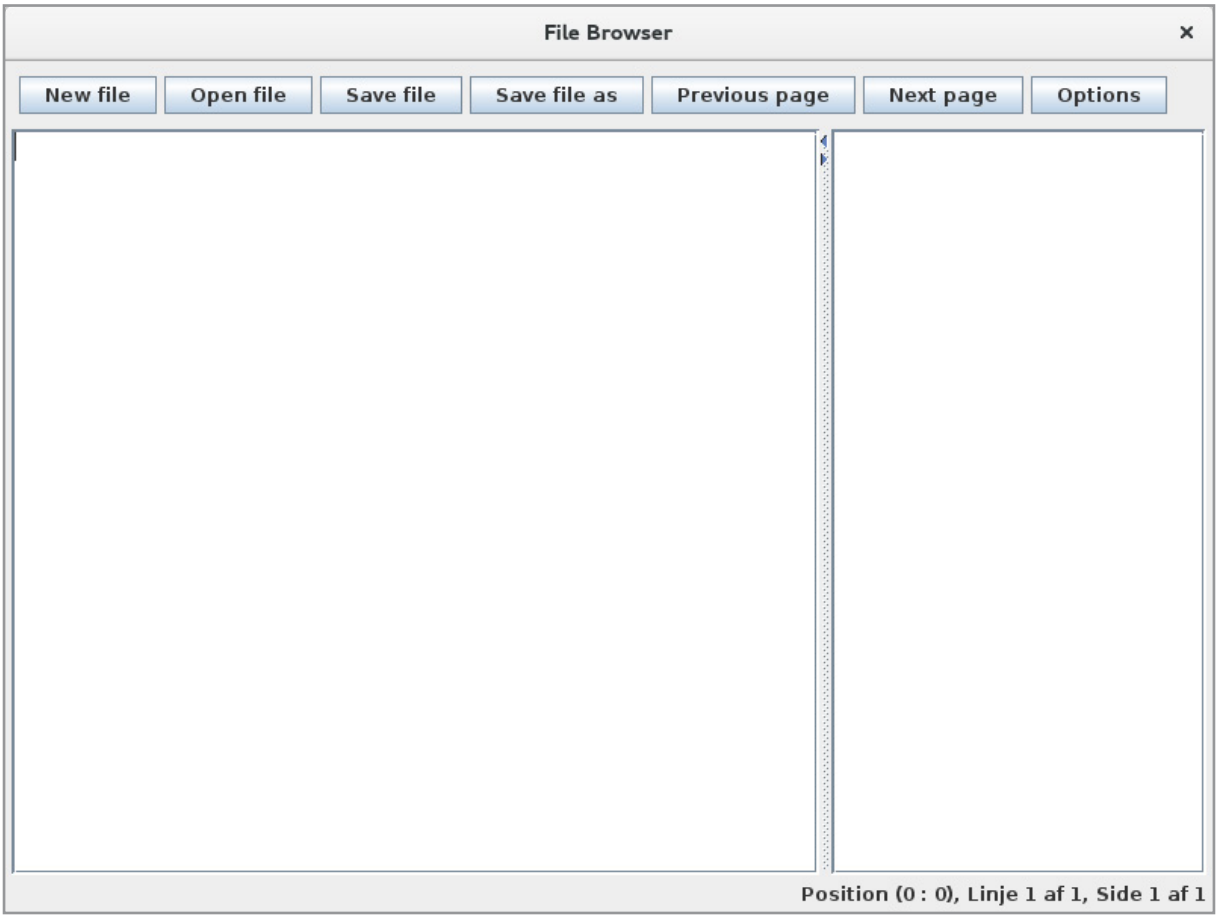
The class *Bitmap* has four instance methods and four static methods that can all raise an exception. You must change the class so all these methods instead raises an exception of type *UtilException*.

Open the test program *TestBitmap* and see if it still can be translated (which it should be). Expand the program with another test method which provoke an exception (as an example an AND of two bitmaps with different length) and test whether you can catch a *UtilException*.

5 FINAL EXAMPLE

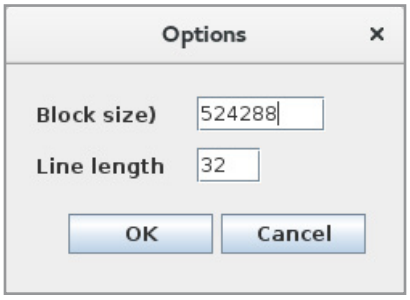
As a final example I will show a program that can open any file and edit the content – whether you have the right to do so. The program should display the content of a file in hexadecimal, and you can edit the file by modifying the individual bytes. The program can edit all files without exception (text files, images, translated programs, etc.), but another question is, whether you can get something out of it, and whether you can interpret the hexadecimal codes.

Contrary to what has been the case with the final examples in the previous books, which have primarily been focused on the process, I will in this example primarily look at the result and hence the program code. I will start with a presentation of the program to explain what the program is doing. When you opens the program, you get the following window:

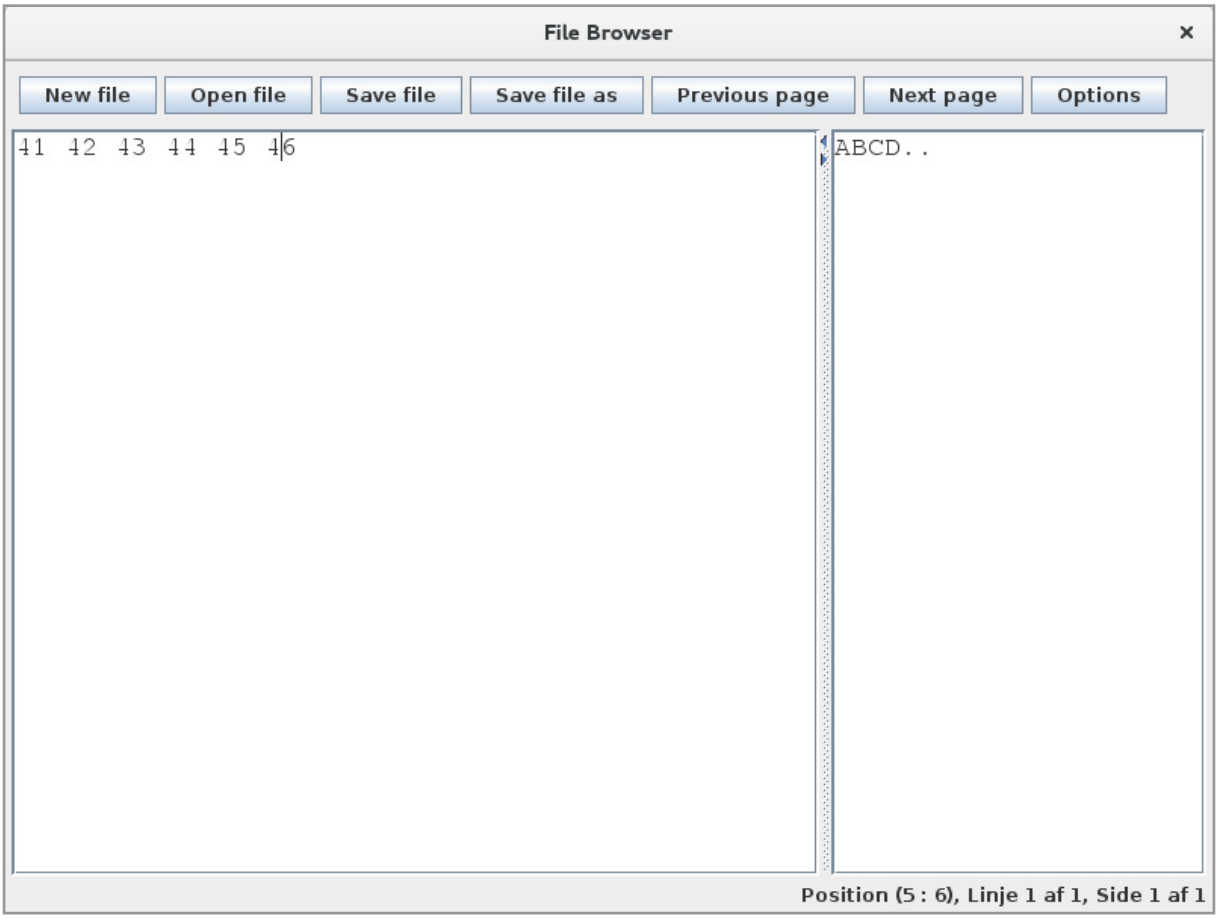


The window contains two *JTextArea* components in a *JSplitPane*. The left component is the one that displays the content of the file in hexadecimal and where you can edit the file. The right component shows the content of the file interpreted as a text – to the extent that it is possible.

The function of the top seven buttons appears largely from the text where the first button clear the window and thus creates an empty file, while the three next are used to respectively open a file, save a file, and save a file as. Because files can be very large, the content of a *JTextArea* can be so large that it is not manageable (the program will be slow), and therefore large files are split into blocks, and the program works with a single block at a time. The fifth and sixth button is used to scroll back and forth in these blocks. Finally opens the last button, a dialog box where you can set the size of the blocks and how many bytes to show per line:



After the program is opened there is not yet loaded any file, but you may well start entering data. Below is the window after I have entered 6 bytes:



Bytes are entered as their hexadecimal codes. Now you can not only enter, but you must insert a byte by pressing the *Insert* key, which inserts a byte at the cursor position width the value 00. Then you can change the value by typing

```
0, 1, 2, …, 9, a, b, …, f
```

The program inserts a space between the individual bytes by itself, and it is only for the sake of readability. If you save the above result in a file and examine the file, you will find that the file contains the text

```
ABCDEF
```

If you look at the program it is in principle a very simple program that alone contains a main window and a single simple dialog box. There is also a model, which, incidentally, is not very complex, and the program consists then of three classes. The most complex is the class M*ainView*, which contains many details.

5.1 THE MODEL

I'll start with the model. The model must represent a data file, and in order to handle large files, that have been defined a simple class that represents a data block to a file:

```
class Block
{
  public byte[] buffer;
  public int size;
```

```
  public Block(int size)
  {
    buffer = new byte[2 * size];
  }
}
```

The class is consisting of a byte buffer and an *int*, which defines the number of bytes in the buffer. The buffer is created by the constructor, where the parameter specifies the buffer's size, but you should note that the buffer is created as twice the size. The reason is that it should be possible to add new bytes, and therefore, there must be some available space. Note that instead I could have applied an *ArrayList<Byte>*, but when I have defined a buffer as an array, it is because you can directly read data to and directly write data from an array when reading and writing files.

Also note that I have deviated from the good principles and defined both variables as *public*. It is to make the code simpler and make references to the class's variables simpler, and when the class has *package* visibility and alone is a helper class for this program, I have allowed not to use the usual data encapsulation. It's actually something that you sometimes see (and there might be good reasons for) with simple helper classes which are only known within the program's package.

The model class is the following:

```
public class Model
{
  private int lineLength = 32;        // number of bytes in a line
  private int blockSize = 524288;     // size og af block in bytes
  private boolean changed = false;    // if content of this block is changed
  private File file = null;           // reference to the file to be edit

  private List<Block> buffers = new ArrayList();
  private int current = 0;
```

By default the program displays 32 bytes in a line, and the block size is as default 524288 bytes (½ megabyte). The last one is a bit of a trade-off, where a small block size means effectiveness in the editing of the file's content, but in return many blocks to be edited one at a time. Increasing the block size also increases the chances that the whole file can be in a single block, but on the other hand, it could be slow to edit the file. These are the two values that can be changed in the *Options* dialog box.

The model also has a variable *changed*, which is used to keep track of the model's on state, and where it is changed corresponding to that the file is edited. Then there is the variable *file* that refers to the file been opened. If the variable is *null*, it means that there is not loaded a file.

The list is for blocks, and by small files (by default less than a ½ megabyte) there is only one block. Along with the list is that the variable *current*, which indicates the block that is currently been displayed in the editor.

The model has several *get* and *set* methods and including methods for navigating the variable *current*, and there is especially more *get* methods that return information about the model's state. They are generally simple, and I will not show these methods here.

The class also has methods to read and save a file. Below is shown the method, which opens a file:

```
public boolean open(File file)
{
  this.file = file;
  buffers.clear();
  try (RandomAccessFile f = new RandomAccessFile(file.getAbsolutePath(), "r"))
  {
    while (true)
    {
      Block block = new Block(blockSize);
      block.size = f.read(block.buffer, 0, blockSize);
      buffers.add(block);
      if (block.size < blockSize) break;
    }
    changed = false;
    return true;
  }
  catch (IOException ex)
  {
    clear();
    return false;
  }
}
```

The method is in principle simple, and the file is read as a *RandomAccessFile*. Input is performed in a loop, which reads a block at a time, and the result is that a large file is divided into several blocks. Similarly is below a method that saved the file:

```
public boolean save()
{
  if (file == null) return false;
  try (RandomAccessFile f = new RandomAccessFile(file.getAbsolutePath(), "rw"))
  {
    long length = 0;
    for (Block block : buffers)
    {
      f.write(block.buffer, 0, block.size);
      length += block.size;
    }
    f.setLength(length);
    return true;
  }
  catch (Exception ex)
  {
    return false;
  }
}
```

This method will overwrite the existing file, and you should especially note that the method ends with updating the file length. It is necessary, as there may be deleted bytes, and the file thus has fewer bytes.

5.2 THE USER INTERFACE

This is essentially the class *MainView*, which is a relatively complex class. Since I do not want to allow the user to edit the content arbitrarily, but only must enter the hexadecimal values of the individual bytes, it is necessary to program the event handling for the keyboard and also the mouse, and it is in fact why the class is a bit complicated. You are encouraged to study the finished code thoroughly, but below I will briefly mention the most important.

The class defines the following variables:

```
private Model model = new Model();
private JTextArea txtEdit;
private JTextArea txtText;
private JLabel lblText = new JLabel();
private JLabel lblPosition = new JLabel();
private JScrollPane scroll1;
private JScrollPane scroll2;
private Scroller scroller;
```

Here are the two *JTextArea* components that are for the views of the current file, where *txtEdit* is used to edit the content as hexadecimal values, while *txtText* is used to display the content interpreted as text. The two *JLabel* components are placed in the bottom of the window and are used to show respectively, the file name (and size), and where in the file the cursor is. The two *JScrollPane* components are for encapsulation of the two *JTextArea* components, and finally the last one whose type is *Scroller*. It is an inner class, which I will not show here, but it must synchronize the two *JScrollPane* containers, such that if you scrolls one then the other scrolls automatically. You are encouraged to study how it works.

As for the design of the user interface there is nothing new, but you should note the method

```
private JTextArea createEditor()
{
  JTextArea txt = new HexEditor();
  txt.setFont(new Font("FreeMono", Font.PLAIN, 18));
  return txt;
}
```

which creates the component *txtEdit*. Its type is *HexEditor*, and it is an inner class that inherits *JTextArea*:

```
class HexEditor extends JTextArea
{
```

```
public HexEditor()
{
  addKeyListener(new KeyHandler());
  setMouseListener();
  setMouseMotionListener();
}
```

The class's constructor adds a new handler for events from the keyboard, and also call the two methods, which remove event handlers for the mouse and define a new one that does nothing but to place the cursor, if you clicks on the component. The handler must solve the problem that you must not place the cursor in front of a space or the end of a line.

As for the keyboard, the program beyond the 16 keys to hexadecimal digits, the program supports the following keys:

- insert
- delete
- the 4 arrow keys
- home and ctrl home
- end and ctrl end
- pagedown
- pageup

wherein the two first modifies the content, while the other navigates the cursor.

The event handler for the keyboard is called

```
public void keyPressed(KeyEvent e)
{
```

and mainly consists of a *switch*, where is switched on the keyboard codes to be treated. Whether it is a key to be treated or not the following statement is executed

```
e.consume();
```

which means that the event is not passed on and thus not to the usual event handling in the base class *JTextArea*. Most entries in the switch statement calls a method that performs the desired action. As an example is shown below the method called, if the insert key is entered:

```
private void insertChar(int p)
{
  if (model.getBlock().buffer.length == model.getBlock().size + 1) return;
  int n = p / 3;
  if (p % 3 > 0) ++n;
  for (int i = model.getBlock().size; i > n; --i)
    model.getBlock().buffer[i] = model.getBlock().buffer[i - 1];
  model.getBlock().buffer[n] = 0;
  ++model.getBlock().size;
  showFile();
  model.change();
  showPosition();
  setCaretPosition(p);
}
```

The parameter is the cursor position. The first thing that happens is that the cursor position must be converted to the byte position in the model, where to insert a new byte, and here one must note that a byte because of the space fills three characters in the component. Here are adopted, if the cursor is in front of a byte, a new byte must be inserted in front, and if the cursor is inside the byte, a new byte must be inserted after. Within the byte can be inserted, the subsequent bytes are moved one place forward and the new byte will have the value 0. Also note that the size of the buffer is counted up by 1. Next is called a method *showFile()* (a method also used in other contexts) which is the method that shows the content of the file (actual the current block). Finally the method *showPosition()* is called, which updates the status bar. These methods are, in principle, simple and not shown here. I will instead show the method that is called when a byte has to be changed:

```
private void changeChar(char ch)
{
  int p = getCaretPosition();
  String text = getText();
  setText(text.substring(0, p) + ch + text.substring(p + 1));
  byte b2 = (byte)(ch >= '0' && ch <= '9' ? ch - '0' : ch - 'a');
  int n = p / 3;
  model.getBlock().buffer[n] = p % 3 == 0 ?
    (byte)(((b2 << 4) | (model.getBlock().buffer[n] & 0x0f)) & 0x000000ff) :
    (byte)(((model.getBlock().buffer[n] & 0xf0) | b2) & 0x000000ff);
  model.change();
  setCaretPosition(p);
}
```

The method's parameter is the character entered, and the first thing that happens is that the component is updated with the character in the right place. Then also the model must be updated. First is determined the character's value as a value between 0 and 15 inclusive. Next, the cursor position again must be converted into the right byte position in the model, and then it is determined whether it is the first or the last of the two digits to be changed. You should note that this method does not call the *showFile(),* what it really should. The reason is performence and the result is that the right component that displays the content as text is not updated, and therefore do not necessarily show the right content.

5.3 THE DIALOG BOX

This leaves the dialog box, which is used to change the parameters of the block size and the number of bytes per line. There is not much to explain, but you should note, that if the dialog box is used, the result is a blank file, and there is not loaded any file. It's a somewhat easy solution, but the reason is to change the block size, it is necessary to create new blocks. You should also note that the coupling between the *MainView* and the dialog box is strong but simple and is also a matter of reaching easy to the target.

APPENDIX A

If you technically have to deal with computers, you can not ignore the binary number, as there are a lot of details that you can not explain without having knowledge of the numbers and their binary representation. The following is a brief introduction to the binary numbers and including the hexadecimal system in view of applications in computer technology, but it is by no means an accurate treatment of number systems in general.

The numbers presented to us at the school are decimal numbers in which numbers are represented by 10 symbols, for example

```
18207
```

When we meet the number, we know exactly what it means, because we know the 10 symbols and their interpretation in relation to where a symbol appears in the number. The decimal number system is a positional number system as the value of a digit is determined by its position:

18207 = 1 tens of thousands plus 8 thousands plus 2 hundreds plus 0 tens plus 7 ones

When we are taught to work with these numbers, we find them simple and easy to understand, but the reason is, that people very easy and very safe can survey and learn the 10 symbols.

The decimal system or 10-number system is not the only number system that people have historically used. For example, was previously used a 60-number system, which you can see the remains of in our division of the time: 1 hour divided into 60 minutes, 1 minute divided into 60 seconds. The 60-number system has also been used in commercially trade calculation. The system has several advantages. 60 has many divisors, which means that you can formulate many rules of arithmetic, and even quite large numbers does not take up so much, but there is however also a very big disadvantage, namely that the system requires 60 symbols (there are actually historical writings that shows the 60 symbols). That the 10-number system that has become the preferred, it is believed that it is because we have 10 fingers. Actually, there are also examples of how people have used a 20-number system (we also have 10 toes).

If you look at integers in the 10-number system, they can be written as $t = \sum_{i=0}^{n} a_i \, 10^i$ where the number has n + 1 digits, and $a_0, a_1, \ldots, a_n$ all is one of the symbols 0, 1, 2, …, 9. As an example is

$$18207 = 7 * 10^0 + 0 * 10^1 + 2 * 10^2 + 8 * 10^3 + 1 * 10^4$$

That is $n = 4$ and $a_0 = 7$, $a_1 = 0$, $a_2 = 2$, $a_3 = 8$ and $a_4 = 1$.

Wee say therefore that the decimal system is a positional number system with base 10. This presentation of the numbers can immediately be generalized to other bases, and actually there is a number system for any natural number greater than 1. As an example the 60-number system mentioned above is a system where the base number is 60.

THE BINARY NUMBER SYSTEM

The binary number system is simply a positional number system with base 2. Thus, it is only necessary with two symbols, and here are used 0 and 1. A number could, for instance be

$$t = \sum_{i=0}^{n} a_i \, 2^i$$

where $a_{0,}a_{1,}\ldots,a_n$ all are 0 or 1. If n = 7 and

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 0 \\ a_2 &= 0 \\ a_3 &= 0 \\ a_4 &= 1 \\ a_5 &= 1 \\ a_6 &= 0 \\ a_7 &= 1 \end{aligned}$$

is

$$t = 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

This number is also written as $t = 10110001_2$ where the last number 2 tells that it is a number from the 2-number system. As you can see, the 2-number system is in principle completly equivalent to the 10-number system, where a number is presented as a sequence of symbols – 10 symbols in 10-number system and 2 symbols in the 2-number system – and the numbers value is determined by the symbols position. The difference is that a symbol is interpreted as a power af 2 instead of a power of 10, and as such is the binary system simpler, since a given power of 2 either is included in the number (the symbol is 1) or it is not (the symbol is 0).

If you look at the above construction, it is easy to determine the value of a number in the 2-number system by a simple calculation of powers of 2:

$$\begin{aligned} t &= 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ &= 128 + 32 + 16 + 1 = 177 \end{aligned}$$

To compare the two systems, please note the following:

$0_2 = 0_{10}$	$100_2 = 4_{10}$	$1000_2 = 8_{10}$	$1100_2 = 12_{10}$
$1_2 = 1_{10}$	$101_2 = 4_{10}$	$1001_2 = 9_{10}$	$1101_2 = 13_{10}$
$10_2 = 2_{10}$	$110_2 = 6_{10}$	$1010_2 = 10_{10}$	$1110_2 = 14_{10}$
$11_2 = 3_{10}$	$111_2 = 7_{10}$	$1011_2 = 11_{10}$	$1111_2 = 15_{10}$

You can thereof immediately see that the numbers in the binary system takes up more space than numbers in the decimal system.

Below are some examples of positive integers written in the 2-number system, and how to convert those numbers to the 10-number system:

$$1110000111110 = 2^{12} + 2^{11} + 2^{10} + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$$
$$= 4096 + 2048 + 1024 + 32 + 16 + 8 + 4 + 2 = 7230$$

$$1111111111 = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 1023$$

$$1110000001 = 512 + 256 + 128 + 1 = 897$$

$$10000000000 = 2^{10} = 1024$$

To determine the value of a binary number – or in other words to convert it to the decimal system – is simple and is just a question to remember the powers of 2. The other way to convert a number in the 10-number system to a binary number requires a little more.

Given a number, for example 2423, one can determine the largest power of 2 that is less than or equal to the number. It is 2^{11} = 2048 and

$$2423 = 2^{11} + 375$$

The largest power of 2 which is less than or equal to 375 is 2^8 = 256. That is

$$2423 = 2^{11} + 375 = 2^{11} + 2^8 + 119$$

The largest power of 2 which is less than or equal to 119 is 2^6 = 64 and

$$2423 = 2^{11} + 375 = 2^{11} + 2^8 + 119 = 2^{11} + 2^8 + 2^6 + 55$$

The largest power of 2 which is less than or equal to 55 is 2^5 = 32 and

$$2423 = 2^{11} + 375 = 2^{11} + 2^8 + 119 = 2^{11} + 2^8 + 2^6 + 55 = 2^{11} + 2^8 + 2^6 + 2^5 + 23$$

To resume, and you will find:

$$2423 = 2^{11} + 2^8 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0 = 100101110111_2$$

That is, to convert a decimal number to a binary number you all the time subtracts the maximum power of 2 and continue until you get 0 or 1.

Below is another example:

$$265 = 2^8 + 9 = 2^8 + 2^3 + 1 = 100001001_2$$

The method is basically simple, but in practice and especially for large numbers, it can be difficult to determine the powers of 2 that you needs. Below is a little more direct way, where you constantly divide by 2:

```
265 : 100100001
132
 66
 33
 16
  8
  4
  2
  1
  0
```

$$265_{10} = 100001001_2$$

You divide by 2 (equivalent to halving the number), and you continues with that until the result is 0. In each division there is a residue that is either 0 or 1. This residue is written after the colon. When finished – quotient is 0 – the bits after the colon is the binary representation in reverse order – that is the most significant bit at the end.

This method is really not so much simpler than the first, the division may be long, but the operations are simple and consists of determine the half of a number and where the remainder is 0 or 1.

```
2423 : 111011101001
1211
 605
 302
 151
  75
  37
  18
   9
   4
   2
   1
   0
```

$$2423_{10} = 100101110111_2$$

Note that it is easy to calculate whether the result is correct.

THE HEXADECIMAL SYSTEM

With reference to the above it is easy to define the hexadecimal system or 16-number system because it is a number system where the base or radix is 16, for example

$$327_{16} = 3 * 16^2 + 2 * 16 + 7 = 807_{10}$$

Again, it is just a question of the individual symbols positions means something different than in the decimal system.

However, there is one problem, since the 16-number system requires 16 symbols, and for this applies the 10 digits and the first 6 letters: A, B, C, D, E and F, and the symbols can then be interpretated as in the following table:

HEX	DEC	BIN	HEX	DEC	BIN	HEX	DEC	BIN	HEX	DEC	BIN
0	0	0000	4	4	0100	8	8	1000	C	12	1100
1	1	0001	5	5	0101	9	9	1001	D	13	1101
2	2	0010	6	6	0110	A	10	1010	E	14	1110
3	3	0011	7	7	0111	B	11	1011	F	15	1111

It is easy to convert a hexadecimal number to the decimal system, and is just a case of simple arithmetic. Below are some examples:

$$1A3E_{16} = 16^3 + 10 * 16^2 + 3 * 16 + 14 = 8718_{10}$$

$$B0032_{16} = 11 * 16^3 + 3 * 16 + 2 = 720946_{10}$$

$$2E_{16} = 2 * 16 + 14 = 46_{10}$$

$$ABCDEF_{16} = 10 * 16^5 + 11 * 16^4 + 12 * 16^3 + 13 * 16^2 + 13 * 16 + 15 = 11292531_{10}$$

The interesting thing about the 16-number system viewed from a computer is that 16 is a power of 2: $16 = 2^4$. There are 16 symbols, and each of these symbols can be precisely expressed binary with 4 bits (see table above). This means that it is extremely simple to convert a hexadecimal number to a binary number, then one simply replaces each hexadecimal symbol with its corresponding binary representation as 4 bits:

$$A3B2_{16} = 1010001110110010_2$$

$$123_{16} = 000100100011_2 = 100100011_2$$

when preceded by 0s has no effect on the value of the number.

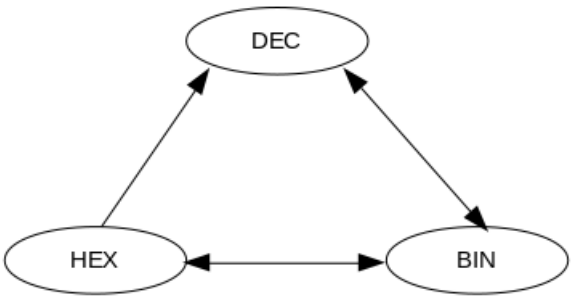
Converting the other way – from binary to hexadecimal – is similarly simple. The binary digits are arranged into groups of 4 bits, and each 4 bits group is converted to the corresponding hexadecimal symbol:

$$1001110101011110_2 = 1001110101011110 = 9D5E_{16}$$

$$1110000101011_2 = 1110000101011 = 1C2B_{16}$$

You can thus conclude that conversion between the binary number system and the hexadecimal system is extremely simple, and in fact it is the same, because the hexadecimal system is merely a shorter representation of the binary numbers. The binary number system is simple – at least seen from a machine, but for us humans, it is difficult to grasp and remember long sequences of 0 and 1. This is where the hexadecimal system comes into play, as it may represent sequences of 0 and 1 in a short way that is much easier for us humans both to read and remember.

Above, I mentioned three number systems and the conversion rules between these systems can be illustrated as follows:



Here you should note that I have not shown any method to convert from decimal to hexadecimal. However, you can immediately use the division method, which has been used when converting from decimal to binary, simply you must divide by 16 instead of 2. In practice it may be a bit difficult to divide by 16 – if it is done manually – and the method has only limited interest. Should you convert a decimal number to hexadecimal, it is easiest to convert the number to binary and from there to a hexadecimal number.

You often hears that a computer work binary, and what you mean by this is, that everything in a computer is represented as binary digits – or perhaps more accurately as sequences of 0 and 1. The reason for that is that it technically is simple to represent two states in a stable and simple manner, for example as two voltage levels, 0 volt may represents 1, while the 5 volts could mean 0. The important thing is that, technically it is easily separate the two voltage levels and in a safe way to determine whether a unit represents 0 or 1.

Internally in a computer the smallest unit that directly can be accessd is a byte, which is a bit pattern consisting of 8 bits. Exactly is the machine's RAM memory a large table of 8 bit patterns. If you have to specify the exact content of such a memory cell in the machine's memory, you specify 8 bits, for example

```
01000001
```

How it is interpreted depends on the program that refers the storage cell, but interpreted it as a binary number, it is the number 65. This number could also be used as a code for the letter A, and a program could thus choose instead of interpreting the above bit pattern as an A.

A computer will always internally represent data as a bit pattern – not necessarily 8 bits, but often 16 or 32, and perhaps even more bits. These bit patterns are difficult to handle for us humans: They are hard to write, and they are hard to remember. This is where the hexadecimal numbers come in, that as the hexadecimal numbers in reality is just a short representation of binary numbers, so they are often chosen to show the binary values for us people in hexadecimal form. The above bit pattern may also be written as hexadecimal

```
41
```

As a further example. Consider the number

28319

If you converts the number to the binary you get:

```
28319 : 111110010111011
14159
 7079
 3539
 1769
  884
  442
  221
  110
   55
   27
   13
    6
    3
    1
    0
```

$$28319_{10} = 110111010011111_2$$

Such a binary number are difficult to remember and hard to convey to others while its hexadecimal representation is much simpler to work with:

$$110111010011111_2 = 110111010011111 = 6E9F_{16}$$

THE INTEGERS

When introduced to the whole numbers in mathematics, you learn that the whole numbers is an infinite set. A computer is a machine and on a machine there is nothing that is infinite, and specifically it means that that a computer can represent only a subset of the whole numbers. As mentioned above, the smallest directly addressable unit is a byte consisting of 8 bits. Since a bit can be either 0 or 1, a byte can represent 2^8 = 256 different bit patterns, and if you only have 1 byte available, it offers 256 different numbers. It is too little, and therefore a machine groups several bytes in a word. If you group 2 bytes, wee say that the unit has a word length of 2 bytes or 16 bits. Similarly, if you group the 4 bytes, you get a word length of 4 bytes, or 32 bits, and the machine is working with 32-bit integers.

Previously, it was common to use 16-bit integer words, in order to have 16 bits available to an integer. As long as you talk about non-negative integers, the last 15 bits are used for the number, while the first is always 0. The number is then represented as a binary number, and as an example is the number 219 represented as:

```
0000000011011011
```

(you can easily calculate that it is the number 219). As another example the number 2890 is represented as:

```
00001011010010 10
```

This representation allows to represents the following non-negative integers:

```
0000000000000000          0
0000000000000001          1
0000000000000010          2
0000000000000011          3
0000000000000100          4
....
0000000011011011        219
....
0111111111111111      32767
```

That is that the largest positive integer is 32767. Note especially that it is the number $2^{15} - 1$, and generally the largest number that can be represented by n bits is $2^n - 1$. The representation of the non-negative integers on a machine is straightforward and the only thing that matters is the word length, which determines how large numbers to work with. Most machines today using a word length of 32 bits for integers, and it means that the largest positive integer is 2^{31} = 2147483647 thus well over two billions. As an example is the number 219 represented as

```
00000000000000000000000011011011
```

on a machine with 32 bit words. You can therefore see the advantages and disadvantages of a large word length. A large word length means the machine can work with big numbers, but should you most of the time only work on small numbers, a large word length result in wasted space, since a large part of the space are just used to 0s.

Sometimes you also uses double-word integers, where you has 64 bits available. The largest positive integer is then 9223372036854775807 whose binary representation is:

```
0111111111111111111111111111111111111111111111111111111111111111
```

And then to the negative numbers, as is represented by their 2 complement. For not having to write too long bit patterns, I will start with a word length of 8 bits, that is a byte. Note that the largest positive integer that can be represented in one byte is

```
01111111 = 127
```

Consider as an example the number 97, and in one byte it is represented as the binary number:

```
01100001
```

By a number's complement wee means the number obtained by inverting all bits, that is,

```
10011110
```

This operation is seen from a machine extremely simple and efficient to implement in hardware and in principle similar to change the voltage level.

A numbers 2 complement is 1 added to its complement and the 2 complement of 97 (= 01100001) is then:

```
10011110
       1
10011111
```

That is the number -97 in an 8-bit unit is represented by the bit pattern

```
10011111
```

It may be noted that adding 1 to a binary number can also be implemented very efficiently in hardware, so the entire operation to determine a number's 2 complement is highly effective.

As another example, consider the number 28, which in a 8-bit unit is represented as

```
00011100
```

Then the representation of -28 is:

```
11100011
       1
11100100
```

This calculation may require a little comment about how to adds two binary numbers. When you adds two decimal numbers, you typical writes the numbers above each other so that they are aligned with the rear digit. The you adds that the digits from back:

1. if the sum of the two digets is les than the base 10, it is the result
2. else you subtract the base 10 from the sum, and the difference is then the result, while you get a carry on 1, to be included in the sum of the next two digits

This method can be applied directly to binary numbers (and in principle on any other number system), and the difference is only that the base number is now 2: The sum at a particular position is either 0 or 1, and is the sum greater than 1, there will be a carry to the next position. All options can be illustrated in the following table:

Bit 1	Bit 2	Carry before	Sum	Carry after
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

It is thus a simple matter manually adding binary numbers.

If you look at the number representation from a computer, it will look at the first bit, and from the value of this bit know if is a negative or a non-negative number, and is the first bit 1, the current software instantly convert the number by take the 2 complement. Actually it is based on the following very important observation that if one determines a number's 2 complement and then again its 2 complement (thus taking the 2 complement twice), then you come back to the original number. For example is the 2 complement of 28

```
11100100
```

and taking the 2 complement to it again, you get:

```
11100100
00011011
       1
00011100
```

that is the number 28.

Consider the number -1. As other negative numbers it is represented by its 2 complement:

```
00000001
11111110
       1
11111111
```

That is where all bits is 1.

Consider the bit pattern

```
10000000
```

that if interpreted as a number of a computer will be interpreted as a negative number:

```
10000000
01111111
       1
10000000
```

and thus the number -128. It means that with a word length of 8 bits, the computer can work with numbers in the interval [-128; 127], which has excatly 256 different integers.

Note that the interval is asymmetric such there is a negative number more than there are positive numbers. The reason is that there must also be room to the number 0.

I will conclude the examination of the negative numbers with a few examples. Consider again number 219, which in a binary 16-bit device is represented as

```
0000000011011011
```

The representation of -219 can be determined as

```
1111111100100100
                1
1111111100100101
```

In a 16-bit device, -1 is represented as

```
0000000000000001
1111111111111110
                1
1111111111111111
```

again only by 1 bits. As the above does not depend on the word length, you may notice that -1 independent of the word length will always be represented as nothing but 1 bits.

Also note what happens if you calculates the 2 complement of 0:

```
0000000000000000
1111111111111111
                1
0000000000000000
```

as just again results in 0.

Consider a 16-bit device that contains the number 1000000000000000, that is where the first bit is 1, but the rest is 0. The machine will interpret this number as a negative number and determine its 2 complement:

```
1000000000000000
0111111111111111
                1
1000000000000000
```

and thus interpret it as -32768. With a word length of 16 bits you can represents the the integers in the range [-32768; 32767].

If the word length is 32 bits, the smallest (negative) number, is

```
10000000000000000000000000000000
```

that is the number -2147483648, and then you get the interval [-2147483648; 2147483647]. With a 64-bit word length the interval is [-9223372036854775808; 9223372036854775807].

If you look at how a modern computer represents negative integers, it's not the only way to do it, and there have also been used other methods. Wee uses as mentioned the 2 complement because it is an operation that can be effectively implemented in hardware, but previously has been used only the complement which is even more effective since it simply just turns the bits. This provides another problem, however. Consider the number 0 in an 8-bit device:

```
00000000
```

If you calculates the number's complement you get

```
11111111
```

that on the machine vould be interpreted as -0, the result, that according to the basic math is 0. This means that when you have two representations of the number 0, there is a problem, which has to be solved in the software, and then complicates the calculations. It is the reason that computers today anywhere using the 2 complement rather than only the complement.

COMPLEMENT ARITHMETIC

The goal of this section is to show how the computer uses the above number representation to calculations. All calculations are performed by the electronics in your computer's processor, and from manufacturers of processors it is about developing processors that are as effective as possible, but also to a competitive in price.

ADDITION

A processor must be able to perform multiple operations, but the main thing is to add two integers. This section will be based on a processor that can perform addition of two binary numbers, and let me begin by are few examples to illustrate what can be done by addition of two binary numbers. Basically, I will count on 8 bits. It is something of a simplification, but since it does not change anything for the basics, it means that I can illustrate it all with fewer bits.

First, recall that with 8 bits, only the interval [-128; 127] are provided.

Calculate 73 + 46

The processor must then adds the two binary numbers 01001001 and 00101110:

```
00010000
01001001
00101110
01110111
```

wherein the top row of bits is the carry. It is easy to figure out that the result is the number 119, which is also the correct result.

Calculate 1 + 2

This is a simple calculation and the result must be 3:

```
00000000
00000001
00000010
00000011
```

Calculate 112 + (-43)

```
112 = 01110000
43 = 00101011
```

The 2 complement of 43:

```
00101011
11010100
       1
11010101
```

-43 = 11010101 and the processor must therefore add the numbers 01110000 and 11010101:

```
11100000
01110000
11010101
01000101
```

This means that you get a result that can quickly be converted to 69, which is the correct result.

Perhaps it is not entirely obvious, for if you manual do the above addition you will see that there is a cary the last time, who apparently just disappears – and actually it do, because there is no room for it. You can say that you drops a sign bit away because the result was positive.

So we can note that a processor may well add a positive and a negative number.

Calculate 43 + (-112)

```
112 = 01110000
43 = 00101011
```

The 2 complement to 112:

```
01110000
10001111
       1
10010000
```

-112 = 1001000 and the processor must therefore add the numbers 1001000 and 00101011:

```
 00000000
 10010000
 00101011
 10111011
```

When the first bit is 1, the result is negative and you have to determine the 2 complement:

```
 10111011
 00000100
        1
 01000101
```

It is the number 69, and then the result is -69, which is also correct.

Calculate 73 + 60

```
73 = 01001001
60 = 00111100
```

The processor must calculate the sum of 01001001 and 00111100:

```
11110000
01001001
00111100
10000101
```

The result is negative, then you have to determine the complement

```
10000101
01111010
       1
01111011
```

Converted to decimal it will be the result -123, which is obviously wrong. What you see here is an example of an overflow: The sum of the two numbers can not be in an 8-bit device and fall outside the range that can be represented with 8 bits. Note that, when the addition of the last two digits there was a cary, which was discarded.

Usually makes a processor nothing in such a situation beyond that it delivers a wrong result. If it is desired to handle this situation, it is up to the software, testing for overflow. If the processor had to do something, it would be more complex, and it would cost in efficiency and it is also not so easy to determine what the processor where appropriate should do. It is not entirely true that the processor is not doing anything. After each addition sets the processor some status bits, and one of them says, there has been an overflow. There is also a status bit, which is the last cary and software can thus test whether there is a cary which is “thrown away”.

It is worth emphasizing that the problem of overflow is not specifically associated with a word length of 8, but it can occur regardless of the word length used. It is only a question of adding two numbers that are sufficiently large.

32 BITS ADDITION

As a final example, I look at addition of two numbers, but this time with a word length of 32 bits, which better reflects what happens in a real processor. I will look at the calculation

2819316 + 32512819

Actually there is nothing new in relation to the above, except that I should write more bits.

First a converting:

```
2819316 =  2B04F4 = 00000000001010110000010011110100
32512819 = 1F01B33 = 00000001111100000001101100110011
```

Then the calculation:

```
00000011110000000011111111100000
00000000001010110000010011110100
00000001111100000001101100110011
00000010000110110010000000100111
```

And so converting the result:

```
00000010000110110010000000100111 = 0000|0010|0001|1011|0010|0000|0010|0111
                                  = 021B2027
                                  = 35332135
```

and it is easy to calculate that the results is correct.

SUBTRACTION

If we now have a machine (a processor) which adds, you also has a machine that can subtract. Should you subtract two numbers, simply adding the second number's 2 complement to the first, and really it's nothing more than to exploit one of the rules in maths:

$$a - b = a + (- b)$$

If we again have a word length of 8 bits, and if you have to calculate 83 – 23 the following happens when you have to calculate the sum 83 + (-23)

```
83 = 01010011
23 = 00010111
```

The complement of 23:

```
00010111
11101000
       1
11101001

-23 = 11101001

10000110
01010011
11101001
00111100
```

The result is 00111100 = 60.

As anothe example: Calculate 2819316 – 32512819, when the word length is 32 bits:

```
 2819316 =  2B04F4 = 00000000001010110000010011110100
32512819 = 1F01B33 = 00000001111100000001101100110011
```

The complement to 32512819:

```
00000001111100000001101100110011
11111110000011111110010011001100
                                1
11111110000011111110010011001101
```

That is

```
-32512819 = 11111110000011111110010011001101
```

Calculation:

```
00000000000111100000100111111000
00000000001010110000010011110100
11111110000011111110010011001101
11111110001110101110100111000001
```

The complement of the result:

```
11111110001110101110100111000001
00000001110001010001011000111110
                                1
00000001110001010001011000111111
```

and the result:

```
00000001110001010001011000111111 = 0000|0001|1100|0101|0001|0110|0011|1111
                                 = 01C5163F
                                 = 29693503
```

The result is as follows, and it is easy to calculate that it is correctly:

```
-29693503
```

The arithmetic as outlined in this section is called *complement arithmetic*, and as you can see, it is sufficient to construct a processor that can add. Then at the same time a processor also can subtract. This is the reasons to implements the representation of negative numbers by their 2 complement.

MULTIPLICATION

If you in the decimal number system multiply a number by 10 (with the base), you must move the number one position to the left and then add a 0. The same is valid for the binary number system: If you have to multiply a number by 2, you must move the bits one position to the left and adding a 0 and such an operation is called a left shift, and it is an operation which is simple to implement in hardware. Consider again the number 219:

```
0000000011011011
```

If you perform a left shift on this number you will get:

```
0000000110110110
```

and a simple calculation shows that it is number 438 corresponding to a left shift is multiplying by 2.

If you look at how to perform multiplication in the 10-number system, for example. 432 * 1322, so this is done by multiplying the 1322 first with 2, then by 3 and finally with 4, but such that you each time shifts 1322 a position left and finally adds to it all:

```
432 * 1322
022100
  2644
 39660
528800
```

The same method can be applied to the binary numbers, but just simpler, because every time either multiply by 0 or 1: To multiply by 0 gives 0, and multiplying by 1 gives the number again.

Suppose you want to multiply 219 by 38, which have the binary values 11011011 and 100110. If the multiplication is carried out with a word length of 16 bits, it may be performed as follows:

```
100110 * 0000000011011011
0000010010000000
0011110110111000
0000000110110110
0000001101101100
0001101101100000
0010000010000010
```

where I have only included the subresults which do not give 0 – there are three such subresults to calculate the sum, as there are three 1 bits in the number 100110. The result is the number 8322.

It is somewhat difficult to perform this multiplication manually, as by adding several binary numbers you need to be more careful to note what you have in carry.

As another example, I calculate the product 752 * 41 in a 16-bit device:

```
752 = 1011110000
41 = 101001

101001 * 0000001011110000
0000100000000000
0011101100000000
0000001011110000
0001011110000000
0101111000000000
0111100001110000
```

that is the number 30832.

In practice, it is not so important to be able to multiply binary, but the important thing is that the multiplication can be performed only by the two operations, left shift and addition and, therefore, can be carried out by a processor which can shift and add.

DIVISION

Division is performed in the decimal number system by subtraction. The division algorithm can also be transferred to the binary number system, and division can then be performed by the processor, which can add.

It is not so easy to divide manually because you have to keep track of whether you have to borrow, but one example.

Consider the number 13556 and assume that it must be divided by 38, and the use of a word length of 16 bits. Note first that the result is an integer, and a simple calculation says that the result should be 356:

```
38 | 13556 | 356
      114..
       215.
       190.
        256
        228
         28
```

The quotient is 356, and you get a remainder of 28, corresponding to the division does not go up.

If you attemptes to copy the usual division algorithm for decimal numbers to the binary system, the method is simpler, since each iteration always results in a 0 time or 1 time.

```
13556 = 11010011110100
38 = 100110
```

The division can now be done binary, as shown below. However, it is not quite easy to do manually:

1. if the number you divide with is greater than the number above, it goes a 0 time, and the upper number goes unchanged on to the next step
2. otherwise it goes a 1 time, and the number you divide by, must then be subtracted from the number above – here it may be necessary to "borrow", which you should keep track of
3. the the next bit is used (as the right bit)
4. continuing until there are no more bits

```
100110 | 0011010011110100 | 00101100100
          100110.........
           011010........
           100110........
            110100........
            100110........
             011101.......
             100110.......
              111011......
              100110......
               101011.....
               100110.....
                001011....
                100110....
                 010110...
                 100110...
                  101101..
                  100110..
                   001110.
                   100110.
                    011100
                    100110
                     11110
```

That is, the result is

```
0000000101100100
```

that is the number 356.

It is not particularly interesting to be able to calculate binary, but the section shows that using the complement arithmetic and a processor that can add it can perform the four arithmetical operations.

BINARY OPERATIONS

Above I have discussed binary arithmetic and explained why the addition is a fundamental operation for a processor. There are other important operations as processors must be able to perform all of which can be implemented efficiently and simple in hardware. This section lists these basic binary operations.

Another reason for looking at these operations is that they are supported by many programming languages, so the languages has operators that correspond to the binary operations. In many contexts, it is important directly to be able to manipulate the individual bits, and to these the binary operators are important.

LEFT SHIFT

This operation I have already mentioned, but given a device with a particular word length the operation shifts all bits one position to the left and insert a 0 in the right end. For example a 16 bits unit

```
0011101111100001
```

and after a left shift the value is:

```
0111011111000010
```

The bit that was left is gone, which is a part of the operation, but in a processor this bit is transferred to a status bit.

If the bit pattern that is shifted to left represents a number it corresponds to, that number is multiplied by 2. Note that this means that if a number is shifted n positions, it corresponds to, that the number has been multiplied by 2^n.

RIGHT SHIFT

A right shift is an operation that shifts all the bits in a word one position to the right and inserts a 0 in first position, for example.

```
0011101111100001
```

and after a right shift the value is

```
0001110111110000
```

If the bit pattern that is shifted, represents a number, it means that the number is divided by 2. If the number is shifted n positions, it corresponds to, that the number is divided by 2^n.

There is a variant of a right shift, known as an *arithmetic right shift*. The difference is that the bit that is inserted in first position is the sign bit. That is, if the first bit is 0, a 0 bit is inserted, and if the first bit is 1, a 1 bit is inserted. If, for example you has the bit pattern

```
0011101111100001
```

is the result of a right arithmetic shift

```
0001110111110000
```

an such the same as a right shift. If, however, the pattern

```
1011101111100001
```

is the result of a right arithmetic shift

```
1101110111110000
```

AND

It is an operation that basically works on two bits, and where the result is a 1 bit if both of the arguments are 1 and otherwise 0. The operation may be expressed in the following table:

Arg 1	Arg 2	AND
0	0	0
0	1	0
1	0	0
1	1	1

The operation corresponds to a conjunction of two statements.

If you have two words an AND of the two words, is a bitwise AND. If, for example you has two 8-bit words:

```
01100010
11110000
```

you determines their AND as follows:

```
    01100010
    11110000
AND 01100000
```

That is, you get 1, where there are two 1 bits, and otherwise the 0.

OR

It is an operation that basically works on two bits, and the result is a bit, which is 0 if both arguments are 0 and otherwise is 1.

The operation may be expressed in the following table:

Arg 1	Arg 2	OR
0	0	0
0	1	1
1	0	1
1	1	1

The operation corresponds to a disjunction of two statements.

If you have two words an OR of the two words is a bitwise OR. If, for example you has two 8-bit words:

```
01100010
11110000
```

you one determines their OR as follows:

```
   01100010
   11110000
OR 11110010
```

That is, you get 0, where there are two 0 bits, and otherwise 1.

XOR

It is an operation that basically works on two bits, and where the result is a bit which is 1 if the two arguments are different and 0 if they are equal. The operation may be expressed in the following table:

Arg 1	Arg 2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

The operation corresponds to the opposite of a biimplication of two statements.

If you have two words an XOR of the two words is a bitwise XOR. If, for example you has two 8-bit words:

```
01100010
11110000
```

you determines their XOR in the following manner:

```
    01100010
    11110000
XOR 10010010
```

NOT

NOT is acting on a single bit by simply turning the bit. The operation can be described in a table as follows:

Arg	NOT
0	1
1	0

The operation is similar to a mathematical negation.

If you have a word a NOT of the word is a bitwise NOT. If, for example you have an 8-bit word:

```
01100010
```

you determines its NOT follows:

```
    01100010
NOT 10011101
```

That is, that a NOT is the same as the complement of the word.

EXAMPLES

I will conclude this section with a few examples of how using the binary operations to manipulate the individual bits in a word. Remember in this context that the smallest unit that directly can be addressed on a computer is a byte, and you can not directly address the individual bits.

Given a 16-bit word:

```
0011101111100001
```

Usually are the bit positions numbered from behind starting with 0, and so that the last bit has index 0, while the first bit has index 15. In this case I would like to set the bit number 3 to 1 – no matter what value it may already have. This can be done with the following expression:

```
0011101111100001 OR 0000000000001000
```

If in a certain position you OR a 0 to the first bit pattern the result in that position will be unchanged and be the same as the value of the first bit pattern – to OR a 0 do not change anything. If in a certain position you OR a 1 to the first bit pattern the result in that position will certainly be 1 regardless of the value of the first bit pattern may have had. The result of the foregoing is, therefore,

```
0011101111100001 OR 0000000000001000 = 0011101111101001
```

The example can immediately be generalized to other word lengths and thus shows how to set a particular bit of 1.

A related task concerns how to set a particular bit to 0, no matter what value it may have.

Given a 16-bit word:

```
0011101111100001
```

the set bit 8 to 0:

```
0011101111100001 AND 1111111011111111
```

If you in a certain position AND a 1 to the first bit pattern the result in that position will be the same as the value of the first bit pattern – that AND a 1 does not change anything. If you in a certain position AND a 0 to the first bit pattern the result in that position certainly will be 0 regardless of the value of the first bit pattern may have had. The result of the foregoing is, therefore,

```
0011101111100001 AND 1111111011111111 = 0011101011100001
```

The example can immediately be generalized to other word lengths and thus show how to set a particular bit to 0.

As a final example I want to show how to test if a particular bit is 0 or 1. Given a 16-bit word:

```
0011101111100001
```

So, I would like to test the value of bit 4. Consider the following expression:

```
(0011101111100001 SHIFT RIGHT 4) AND 0000000000000001
```

If the value of this expression is 0, then the bit 4 is also 0. Otherwise, bit 4 is 1. You could also also test bit 4 more directly by looking at the value of the following expression:

0011101111100001 AND 0000000000010000

ENCODING OF CHARACTERS

Above I have shown how to represent integers on a computer that works exclusively binary – that is where all data are represented as sequences of 0 and 1 bits. I have also demonstrated how the computer by the use of complement arithmetic can perform the four arithmetical operations. A computer works with other data types than integers and such a computer must be able to work with text. In this section I will show how to represent text using binary numbers.

There are several ways, but the principle is simple, as you for each letter, digit and other characters associates a numeric code, and the only thing that is necessary is to agree on an encoding table that for each character determines which code to be used. That is where the differences occurs, as historically are used multiple tables.

ASCII

I'll start with a table called ASCII (*American Standard Code for Information Interchange*) that may not be used directly with modern computers, but it is yet an important table to know. The table encodes characters as 1 byte numeric codes, and the table then has room for 256 characters. The table is usually divided into three parts

1. the codes 0–31 that defines various control characters
2. the codes 32–127 that defines standard characters from the english alphabet
3. the codes 128–255 that defines among other country-specific characters

Note that the first two parts totally consists of 128 codes, and thus can be represented by 7 bits, and the first ASCII tables (the first standard) covered only the first two parts and was thus a 7-bit encoding.

Below is the first part of the table where I partly have shown codes (both in decimal, binary and hexadecimal) and partly symbolic character's name and a brief descriptive text. Note that these codes do not directly correspond to the characters on the keyboard but are control codes, many of which are defined for the purpose of data communication in which text has to be transmitted over one or another communication line.

DEC	BIN	HEX	Symbol	Text
0	00000000	00	NUL	Null char
1	00000001	01	SOH	Start of heading
2	00000010	02	STX	Start of text
3	00000011	03	ETX	End of text
4	00000100	04	EOT	End of transmission
5	00000101	05	ENQ	Enquiry
6	00000110	06	ACK	Acknowledgment
7	00000111	07	BEL	Bell

DEC	BIN	HEX	Symbol	Text
8	00001000	08	BS	Back space
9	00001001	09	HT	Horizontal tab
10	00001010	0A	LF	Line feed
11	00001011	0B	VT	Vertical tab
12	00001100	0C	FF	Form feed
13	00001101	0D	CR	Carrige return
14	00001110	0E	SO	Shift out / X-on
15	00001111	0F	SI	Shift In / X-off
16	00010000	10	DLE	Data link escape
17	00010001	11	DC1	Device control 1 (oft. XON)
18	00010010	12	DC2	Device control 2
19	00010011	13	DC3	Device control 3 (oft. XOFF)
20	00010100	14	DC4	Device control 4
21	00010101	15	NAK	Negative acknowledgement
22	00010110	16	SYN	Synchronous idle
23	00010111	17	ETB	End of transmitblock
24	00011000	18	CAN	Cancel
25	00011001	19	EM	End of medium
26	00011010	1A	SUB	Substitute
27	00011011	1B	ESC	Escape
28	00011100	1C	FS	File separator
29	00011101	1D	GS	Group separator
30	00011110	1E	RS	Record separator
31	00011111	1F	US	Unit separator

Below I have listed the codes used with plain text:

- LF, used for line break
- HT, used for the tabulator character
- FF, used for page break
- ESC, the ESC key
- BS, the backspace key

Below is the second part of the table, which defines codes for characters in the English alphabet:

DEC	BIN	HEX		DEC	BIN	HEX		DEC	BIN	HEX	
32	00100000	20		64	01000000	40	@	96	01100000	60	´
33	00100001	21	!	65	01000001	41	A	97	01100001	61	a
34	00100010	22	"	66	01000010	42	B	98	01100010	62	b
35	00100011	23	#	67	01000011	43	C	99	01100011	63	c
36	00100100	24	$	68	01000100	44	D	100	01100100	64	d
37	00100101	25	%	69	01000101	45	E	101	01100101	65	e
38	00100110	26	&	70	01000110	46	F	102	01100110	66	f
39	00100111	27	'	71	01000111	47	G	103	01100111	67	g
40	00101000	28	(	72	01001000	48	H	104	01101000	68	h
41	00101001	29	)	73	01001001	49	I	105	01101001	69	i
42	00101010	2A	*	74	01001010	4A	J	106	01101010	6A	j
43	00101011	2B	+	75	01001011	4B	K	107	01101011	6B	k
44	00101100	2C	,	76	01001100	4C	L	108	01101100	6C	l
45	00101101	2D	-	77	01001101	4D	M	109	01101101	6D	m
46	00101110	2E	.	78	01001110	4E	N	110	01101110	6E	n
47	00101111	2F	/	79	01001111	4F	O	111	01101111	6F	o
48	00110000	30	0	80	01010000	50	P	112	01110000	70	p
49	00110001	31	1	81	01010001	51	Q	113	01110001	71	q
50	00110010	32	2	82	01010010	52	R	114	01110010	72	r
51	00110011	33	3	83	01010011	53	S	115	01110011	73	s
52	00110100	34	4	84	01010100	54	T	116	01110100	74	t
53	00110101	35	5	85	01010101	55	U	117	01110101	75	u
54	00110110	36	6	86	01010110	56	V	118	01110110	76	v
55	00110111	37	7	87	01010111	57	W	119	01110111	77	w

DEC	BIN	HEX		DEC	BIN	HEX		DEC	BIN	HEX	
56	00111000	38	8	88	01011000	58	X	120	01111000	78	x
57	00111001	39	9	89	01011001	59	Y	121	01111001	79	y
58	00111010	3A	:	90	01011010	5A	Z	122	01111010	7A	z
59	00111011	3B	;	91	01011011	5B	[	123	01111011	7B	{
60	00111100	3C	<	92	01011100	5C	\	124	01111100	7C	|
61	00111101	3D	=	93	01011101	5D	]	125	01111101	7D	}
62	00111110	3E	>	94	01011110	5E	^	126	01111110	7E	~
63	00111111	3F	?	95	01011111	5F	_	127	01111111	7F	

Note particularly the code 32, which is a space.

This encoding is not sufficient, as there are a number of other letters that are national. For example the Danish letters æ, ø and å, but there are also other characters found on a keyboard, for example ½, and may also include other special characters such as the Greek alphabet.

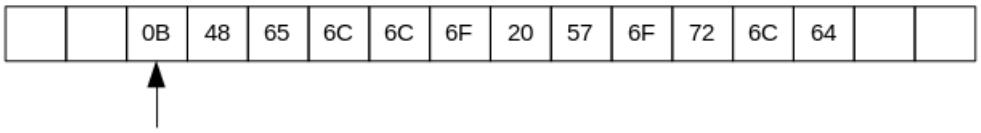
Encoding of these symbols have been solved by defining extended ASCII tables for codes 128–255 that defines additional 128 characters. A concrete machine can then load the table extension that you want to use. As an example is a code table, called ISO 8859-1, which includes the Danish letters, and below is shown the codes of the Danish letters in ISO 8859-1:

- æ 230
- ø 248
- å 229
- Æ 198
- Ø 216
- Å 197

Consider as an example the text

Hello World

where there are 11 characters. If this text is encoded as ASCII and stored in the machine's memory, it uses 11 bytes plus one, and the contents of the memory might be:



where the content is shown in hexadecimal (so it's easier for us humans to read), and where the arrow indicates the address where the text is stored. Note the first byte that indicates how long the text is. In one way or another there must be stored information about where the text ends, and it does not have to be a counter, as shown above, but it is one of the ways used. Note also that if you as illustrated above uses one byte to specify the length, then a text has a maximum length of 255 characters. The problem can of course easily be overcomed by using two bytes to or more to indicate the length.

It is important to note that there nowhere are said something about that the 12 bytes should be interpreted as text, and how those bytes are processed is completely determined of the program that reads the 12 bytes. As an example could a program choose to interpret the 12 bytes as three 32-bit integers, and the program will then read the following numbers:

```
0B48656C = 189293932
6C6F2057 = 1819222103
6F726C64 = 1869769828
```

which is something completely different than the text *Hello World*. A program can in principle read arbitrary bytes and interpret them as it will, but if it makes sense is a whole different matter.

UNICODE

The encoding used today, is called *Unicode*, and the purpose is to replace the many different code pages known from ASCII, with a single code page, which contains all possible characters in all different languages. The price for it is obviously a very large table (just think of all non-European languages, chemical symbols, mathematical symbols, etc.), and it is also the system's disadvantage, both because it can be difficult to define a table, which all agree on, or that it could be "expensive" to use such a large table, if you really only need a small subset.

There is defined two different systems:

- UTF (Unicode Transfer Format)
- USC (Universal Character Set)

and each of these systems has multiple encodings, and I will mention the following:

- UTF-8, which is an 8-bit variable length encoding, which basically is a kind of extension of the ASCII system, and in particular is widely used on the Internet
- UTF-16 which is a 16 bit variable length encoding used by Windows
- UTF-32, which is a 32-bit fixed-length encoding, is used to some extent in the Unix and Linux (the same as the UCS-4)

Mention must also UCS-2, which is a fixed-length 16-bit encoding, that only has support a subset of all Unicodes.

That an encoding is of fixed length means that all characters take up the same. In UTF-32 all the characters so fills 4 bytes. Is an encoding of variable length, the characters does not fill the same, and basically it is the principle that characters used often has codes with a smaller size, while characters used rarely take up more space. It is a choice. Fixed length encoding is the simplest but pay by space consumption, while a variable length encoding minimizing the required to space, and pay with a more complex encoding, which places extra demands on the software.

UTF-16

I'll start with UTF-16 as the encoding used under Windows and also Linux. All codes are 2 or 4 bytes. The first 256 ASCII codes, that fully correspond to ISO 8859-1, uses two bytes (16 bits) for each character. As an example is the encoding of a large A

```
0041
```

For all of the most frequently occurring characters 16 bits are used and they have thus hex codes from 0000 to FFFF, which gives 65,536 different codes. All other codes occupies 4 bytes or 32 bits. All the 16 bits codes is called *Basic Multilingual Plane* (BMP).

UTF-16 is an ISO standard that has the name ISO / IEC 10646 and include most of the world's characters. In principle, the 4 bytes to assign codes corresponds to to 4,294,967,296 characters, but not all are used, and the encoding of 4 bytes code is relatively complex. Below is a general description.

For codes outside the BMP defines UTF-16 character codes in the range 10000 – 10FFFF that gives 1048576 codes. Each code is translated into two codes, called a surrogate pair. The procedure is as follows:

1. if the code, which according to the standard belongs to the interval [10,000; 10FFFF] subtracts 10000 from the code, and the result is certainly less than or equal to FFFFF, and thus fills a maximum of 20 bits
2. split the 20-bit code into 2 halves of 10 bits
3. adds D800 to the left half (as max is 3FF) that gives a value in the range [D800; DBFF]
4. adds DC00 to the right half (as max is 3FF) that gives a value in the range [DC00; DFFF]

It provides precise 1024 * 1024 = 1048576 pairs (10 bits provides 1024 different values).

The standard guarantees that there will never be defined surrogate pairs outside these ranges. Note that the two intervals are disjoint, and the reason for the somewhat complex encoding is that it must be simple to decode four bytes unicodes.

UTF-8

UTF-8 is an encoding that is used much on the Internet as it directly is an expansion of ASCII coding, and as it strives to be sent as few bits as possible. It is a variable-length encoding, using from 1 to 6 bytes per character. In practice, however, a maximum of 4 bytes, as it is sufficient to encode the entire UTF-16 area.

The table below shows the principles for encoding the UTF-16 area, where a b indicates a data bit:

Unicode	UTF-8	Number of characters
000000 – 00007F	0bbbbbbb	128
000080 – 0007FF	110bbbbb 10bbbbbb	1920
000800 – 00FFFF	1110bbbb 10bbbbbb 10bbbbbb	63488
010000 – 1FFFFF	11110bbb 10bbbbbb 10bbbbbb 10bbbbbb	2031616

UTF-32

This encoding uses 32 bits for all the characters, and that means that text encoded by this standard takes up more memory. The system is used only to a limited extent in the Unix world. In principle, many text operations are simpler when all characters take up the same, but in practice the gains are modest.

REPRESENTATION OF DECIMAL NUMBERS

Above I have shown how to represent integers in a computer where positive integers are directly represented as binary numbers, and negative integers are represented by their 2 complement. A computer must also be able to work with decimal fractions, and it is once more difficult. There are a number of ways, but basically they can be divided into two categories

1. decimal numbers with fixed decimal point, which in many ways is an extension of the representation of integers, so that there is a number of the bits for the integer part and a number of bits for the fractional part
2. decimal numbers with floating point, where you have a certain number of bits for the whole number, but the bits used for the integer part and which are used for the fractional part depends on the numbers current value

The two categories are quite different, and the first is by far the simplest, but is best suited if the numbers do not spread over too wide an interval. The second category enables a far greater range of numbers, but the representation is more complex.

I will only look at the last category. Partly because it is the most used form of representation of the decimal numbers, and partly because that's where most are to add in terms of what I have previously shown. One speaks generally about floating-point numbers, and here again there are several variants, but I would look at a standard called *IEEE Standard 754*, which is the most commonly used standard for representing floating-point numbers.

A BIT MORE ABOUT BINARY NUMBERS

In the introduction I mentioned that an integer in the 10-number system can be represented as a sum

$$\sum_{i=0}^{n} a_i \, 10^i$$

where the symbols a_0, a_1,…a_n are digits in the decimal number system, and I also mentioned how this representation can be generalized to an arbitrary base and especially to the base 2 and the binary number system. Looking again at the 10-number system, the notation can be used to describe arbitrary decimal numbers, that is numbers which include fractions:

$$\sum_{i=-m}^{n} a_i \, 10^i$$

If you looks at the number 1234.56, it can be written as

$$\sum_{i=-2}^{3} a_i \, 10^i$$

where $a_{-2} = 6, a_{-1} = 5, a_0 = 4, a_1 = 3, a_2 = 2, a_3 = 1$.

This notation in which you works with negative powers of the basic number, may also be used for binary numbers. Consider the binary number

```
10110.10010111
```

which this time has a fraction. It is easy to determine the number's value as a decimal number, because it still just is a sum of powers of 2:

$$10110.10010111_2 = 2^4 + 2^2 + 2 + 2^{-1} + 2^{-4} + 2^{-6}2^{-7} + 2^{-8} = 22.58984375$$

However, you should note that if the result is calculated on a machine, the result will be rounded, if there are many bits after the decimal point, for example

$$1.00000000000011_2 = 1 + 2^{-13} + 2^{-14} = 1.0001831054688$$

You can also convert the decimal number to a binary fraction. This is done by converting the integer part and the fractional part separately. Consider the number 123.45

The integer part can be converted as shown previously:

```
123 =
64 + 59 =
64 + 32 + 27 =
64 + 32 + 16 + 11 =
64 + 32 + 16 + 8 + 3 =
64 + 32 + 16 + 8 + 2 + 1
```

The fraction part can be converted by multiplying by 2:

- multiply the fraction part by 2
- the integer part, that is 0 or 1, is the next bit
- repeat above until the fraction part is 0, or you have the wanted number of bits

This means that you finds the next bit of multiplying by two, and then continue with the fractional part to this result.

In this case, the method is:

```
   45
0  90
1  80
1  60
1  20
0  40
0  80
1  60
....
```

and the result is that `1111011.0111001 = 123.45` which is a rounded result.

As another example, the consider the decimal number 0.0825. Here, it is only necessary to convert the fractional part, as the integer part is 0:

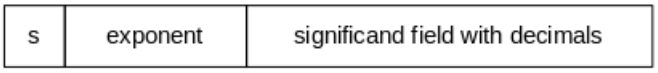
```
  0825
0 1650
0 3300
0 6600
1 3200
0 6400
1 2800
0 5600
1 1200
0 2400
0 4800
0 9600
1 9200
1 8400
1 6800
1 3600
```

The result: `0.0825 = 0001010100011111`

Note that if I converts the binary result to a decimal number, I get 0.0824890137 so you can see that there must be many bits to get an accurate result.

FLOITING POINT

The decimal numbers are a subset of the real numbers, and the problem is to represent real numbers using a finite number of bits, for example 32 or 64 which is the most common. In general, a real number is represented in the form



where the *s* field always fills 1 bit and represents the sign, the exponent field is an exponent and the significand field is used for number's digits.

The main problem is that the representation of a real number is not unique, but is always a rounded result. A representation of the integers defines a subset of the integers, such that all integers in a range is represented. Similarly it is, by the representation of the real numbers, that since there are only a limited number of bits available, wee can represent only a subset of the real numbers, and thus the real numbers within a range, but unlike the integers includes any interval of real numbers infinite many numbers, and any representation of real numbers by using a specific number of bits can only represent a subset of the range. Another problem is that a decimal number such as 123.45 can not be converted to a binary number with a finite number of bits:

```
123.45 = 1111011.011100110011001100110011……
```

So there is no clear relation between decimal fractions and the binary numbers.

32 BITS FLOITING POINTS

I'll start with a representation that takes up 4 bytes, i.e. 32 bits:

- 1 bit to the sign
- 8 bits for the eksponent
- 23 bits for the significand

The sign bit is 0 if the number is non-negative and 1 if the number is negative. If you look at the significand field it contains the 23 bits:

bbbbbbbbbbbbbbbbbbbbbbb

and it is interpreted as the binary number

$$1.bbbbbbbbbbbbbbbbbbbbbbb * 2^{n-127}$$

where n is the value of the exponent field. 127 is called the *bias value*, which is binary 01111111. It is used to ensure that the content of the exponent field is not negative. I would as an example look at the number 123.45. Since it is a positive number is the sign bit 0. The number's binary representation is

$$123.45 = 1.11101101110011001100110011001100110011001100110011\ldots * 2^{6}$$

Therefore, the content of the significand field is `11101101110011001100110` (the first 1 digit before the decimal point is implicit). Since 133-127 = 6 is the content of the exponent field is 133 or binary `10000101`. Then the number 123.45 is represented as the bit pattern

```
01000010111101101110011001100110
```

In the case of floiting point are not used complement arithmetic, and a negative number is handled only by changing the sign bit. As an example is -123.45 is represented as

```
11000010111101101110011001100110
```

that is the same bit pattern as 123.45 except the sign bit.

AN EXAMPLE

To highlight the principle I will look at a small calculation. Suppose a floating point contains the following bits:

```
11000111001100111100111001010100
```

When the first bit is 1, it represents a negative number. The exponent part is `10001110`, which is the number 142, and subtract the bias you gets 15.

The significand part is `01100111100111001010100` and the number is then

$$1.01100111100111001010100 * 2^{15} = 1011001111001110.01010100 = 46030.3213$$

and the value of the number is -46030.32813.

There is little variation in terms of how the boundary conditions are treated, but basically can be assumed as follows. The exponent field represents values in the range of 0 to 255, but 255 which has the value 11111111 that is illegal. The power of 2 exponent thus lies between -127 and 127. The largest positive number is:

```
01111111011111111111111111111111
```

The exponent is 11111110 – 127 = 254-127 = 127, and the greatest positive number is therefore

$$1.11111111111111111111111 * 2^{127} = 11111111111111111111111 * 2^{104}$$
$$= 3.402823466 * 10^{38}$$

Similarly, the smallest positive number:

```
00000000000000000000000000000001
```

The exponent is 00000000 – 127 = -127 and the number is

$$1.00000000000000000000001 * 2^{-127} = 10000000000000000000001 * 2^{-150}$$
$$= 5.87747245 * 10^{-39}$$

(it is one of the numbers that vary slightly). The result is that you can assume that the standard may represents numbers within the following range:

$$[-10^{38};\ -10^{-38}] \cup [10^{-38};\ 10^{38}]$$

and that when 2^{24} = 16777216, there are about 7 significant digits.

Individual bit patterns are assigned a special meaning:

Bit pattern Value

```
00000000000000000000000000000000    0
01111111100000000000000000000000    Infinity
11111111100000000000000000000000    Minus inifinity
01111111110000000000000000000000    Not a number
```

but there are others.

64 BITS FLOITING POINTS

The principle is the same as above, but using 64 bits which are interpreted as follows:

- 1 bit to the sign
- 11 bits for the eksponent
- 52 bits for the significand

```
The bias value is 1023, that binary is 01111111111.
```

Consider as an example the number 12345.6789. Binary it is

```
11000000111001.1010110111001100011000111111000101000001.....
```

or

$$1.1000000111001101011011100110001100011111100010100001.....*2^{13}$$

Since 13 = 1036 – 1023 and 1036 = 10000001100 is the representation of 12345.6789:

0100000011001000000111001101011011100110001100011111100010100001

Slightly rounded it provides the opportunity to represent numbers within the following range:

$$[-10^{308};\ -10^{-308}] \cup [10^{-308};\ 10^{308}]$$

and it means approximately 15 significant digits.

ANOTHER EXAMPLE

As a final calculation. Assume that a real number is represented as the following 64-bit pattern

```
0100000100010111000101101100100111001111110001111010001110011000
```

Sign:

0, the number is positive

Exponent:

```
10000010001 - 1023 = 1041 - 1023 = 18
```

Significand:

```
01110001011011001001110011111100011111010001110011000
1.01110001011011001001110011111100011111010001110011000 * 2^18 =
101110001011011001 0.01110011111100011111010001110011000 =
378290.45291
```