Poul Klausen

JAVA 4
Java's type system and collection classes

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 4: JAVA'S TYPE SYSTEM AND COLLECTION CLASSES

SOFTWARE DEVELOPMENT

Java 4: Java's type system and collection classes: Software Development
1st edition
© 2017 Poul Klausen & bookboon.com
ISBN 978-87-XXXX-XXX-X
Peer review by Ove Thomsen, EA Dania

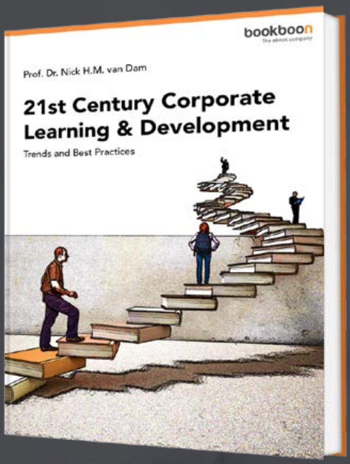
CONTENTS

	Foreword	**7**
1	**Introduction**	**9**
2	**Wrapper classes**	**11**
	Exercise 1	14
3	**Strings**	**16**
3.1	StringBuilder	18
3.2	StringTokenizer	19
	Problem 1	20
3.3	Regular expressions	23
	Exercise 2	36

Free eBook on Learning & Development

By the Chief Learning Officer of McKinsey

Download Now


bookboon
Prof. Dr. Nick H.M. van Dam
21st Century Corporate Learning & Development
Trends and Best Practices

4 Inner classes 38
4.1 Iterators 40
Exercise 3 47
Exercise 4 47
4.2 Example: ZipCodes 48

5 Enumerations 54
Exercise 5 58
Problem 2 59

6 Exception handling 70
6.1 Checked exceptions 74
6.2 Unchecked exceptions 81
Exercise 6 83

7 Generic types 85
Exercise 7 97
7.1 More on parameters 98
Exercise 8 99
7.2 Raw classes 100
7.3 Generic methods 101
Exercise 9 102
Problem 3 104
7.4 Bound parameter types 107
Exercise 10 110
7.5 Generic types and inheritance 111

8 Lambda expressions 115
8.1 Anonymous classes 115
8.2 Methods as parameters 119
8.3 Examples of lambda expressions 122
8.4 Java functional interfaces 129
8.5 Event handlers 129

9 Collection classes 131
9.1 Overview of the collection classes 131
9.2 ArrayList 134
9.3 LinkedList 138
Exercise 10 144
Problem 3 144
9.4 HashSet 145
9.5 TreeSet 149
Exercise 11 150
9.6 HashMap and TreeMap 153
9.7 PriorityQueue 155
Exercise 12 155
9.8 The algoritms 157
Exercise 12 161

10 Annotation 162

11 Packages 164

12 Final example 166
12.1 The task 166
12.2 Analysis 167
12.3 Design 170
12.4 Programming 174
12.5 Test 177
12.6 The last step 178

FOREWORD

This book is the fourth in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. This book has, however, only to a lesser extent focus on the process, but more on the language and numerous of details regarding Java as an object-oriented programming language. The book is thus primarily for the programmer and presents techniques that can help to ensure the development of robust and maintenance-friendly programs, but also techniques needed to know in order to develop programs in a modern programming language. You can also say that the current book deals with details on concepts, you have met in the previous books, but only have been touched without going in depth. The book is a natural continuation of the book Java 3 on object-oriented programming and thus assumes that the reader has a knowledge corresponding to what is addressed in Java 3.

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

In Java 1 and Java 3, I have relatively detailed treated Java's types, basically divided into value types and reference types. For reasons of practical programming you can go on with what has already been said about types, but Java defines a lot more, and it is the subject of this book. It's kind of concepts, which purpose are to develop programs of better quality, and also to write programs with less code. Part of the following concepts I have already used several times, so the book also serves as an explanation of the concepts that I previously have used without exactly explaining what happens.

The main concepts are

- Wrapper classes
- Strings
- Inner classes
- Exception handling
- Generic methods and classes
- Lambda expressions
- Collection klasser

Most of what follows is something that you can live without, but conversely something that can make life easier as a programmer, but also kan help to increase the quality of the programs developed.

The title of this series of books is software development, and the title suggests, I want to focus on the development process and to a lesser extent the Java programming language. This is not so in this book, as there is largely talk about details of the language itself, but also concepts that are part of any modern programming language.

Here it is worth thinking about that Java from the start was designed to should be an object-oriented language that was simple and easy to learn. Java has been a success and has over time progressed much as the language is in use in more and more areas from both the development of complex computer applications over the web applications to the development of apps for mobile phones. This of course has caused the development of a number of new APIs, each of which aims to support the development of a specific category of applications or support a particular technology. At the same time the basic concepts of the language are developed, really as a natural evolution of what has happened with other programming languages, and where there are more and more concepts added into the language that is many of the concepts discussed in this book. In principle, it is fine as it helps software developers, but conversely there is a price, namely that the language becomes more complex and difficult to learn, and it is indeed a significant step away from what was originally the idea of Java. It is worth thinking about, because in the worst case it could mean that the language die because of its own success, and in fact there are several precedents of programming languages that has had wide circulation, which has constantly evolved to finally departing at death because they were too complex and hopeless to learn. Everything in life is a balance.

2 WRAPPER CLASSES

Java's type system consists as discussed in the previous book of a hierarchy of classes with the class *Object* as the root. Beside there is the simple types primarily for numbers and characters, and in some places it is a problem that these are not part of the object-oriented class hierarchy. There are, therefore, for each of the simple types defined a class that encapsulates the simple type. For example is the class *Integer* a class that encapsulates an *int*. These classes are called wrapper classes, and besides they allow the variables of the simple types to be used in the same way as other objects, the wrapper classes defines a number of useful methods and constants for the simple types.

For for the numeric types, there are following wrapper classes

- *Byte*
- *Short*
- *Integer*
- *Long*
- *Float*
- *Double*

and the names should tell the primitive type as the class encapsulates. The six classes are all derived from the class *Number*.

There are also wrapper classes to *char* and *boolean* and they are called respectively *Character* and *Boolean*. Generally are the use of these classes without major challenges, but you should study them well including to be aware of the methods they offer. As an example is shown a method which uses some of the wrapper classes:

```
private static void test01()
{
 Integer a = new Integer(23);
 Integer b = 23;
 print(a);
 print(b);
 Double x = 3.14;
 print(x);
 Character c = 'A';
 print(c);
 print(19);
}
```

```
private static void print(Object obj)
{
 System.out.println(obj.getClass().getName() + " " + obj);
}
```

The method *print()* prints an *Object* as the name of its class as well as its value. The variable *a* is an *Integer* object with the value 23, and you will notice that it is created with *new* as other objects. *b* is also an *Integer* object, but is assigned the *int* value 23 directly. Where it is legal, it is because the compiler uses a concept called auto boxing. This means that when the compiler sees that the variable *b* is of the type *Integer*, and you try to assign a numerical value, the compiler knows well that this value should be encapsulated in an *Integer* object, and it will automatically execute *b = new Integer(23)*. The same applies for variables *x* and *c*, using the auto boxing to respectively a *Double* and a *Character*.

Note especially the last statement. The compiler will look for a *print()* method with an *int* as a parameter. Such does not exist, but there is a *print()* method with an *Object* as parameter, and the compiler will automatically boxes the value 19 in an *Integer*. Is the method performed, you get the result:

```
java.lang.Integer 23
java.lang.Integer 23
java.lang.Double 3.14
java.lang.Character A
java.lang.Integer 19
```

The following method creates an *ArrayList* of objects of the type *Number* and adds four items to the list:

```
private static void test02()
{
 ArrayList<Number> list = new ArrayList();
 list.add(2);
 list.add(3.14);
 list.add((short)5);
 list.add(7L);
 for (Number t : list) print(t);
}
```

Note especially that when the list is created, it is not necessary (but legally) to write:

```
ArrayList<Number> list = new ArrayList<Number>();
```

The compiler knows that *list* should be used for objects of the type *Number*. Also note that the four objects that are added to the list has a different type. If the method is executed is the result:

```
java.lang.Integer 2
java.lang.Double 3.14
java.lang.Short 5
java.lang.Long 7
```

Consider as the last example the following method

```
private static void test03()
{
 Integer[] arr = { 2, 3, 5, 7 };
 int s = 0;
 for (Integer t : arr) s += t;
 System.out.println(s);
}
```

Do you want to make a difference?
Join the IT company that works hard to make life easier.
www.tieto.fi/careers
Knowledge. Passion. Results.
tieto

Here is *arr* an array of the type *Integer*, which is initialized with four objects. Note that the compiler performes auto boxing. Next, a loop which determines the sum of the elements of the array. Here is *s* an ordinary *int* variable, and you ought really to write:

```
s += t.intValue();
```

but it is not necessary because the compiler use *automatic unboxing*. The compiler sees that *s* is an *int* and know that it must use the value of the *Integer* object *t*.

EXERCISE 1

In this exercise you has to test the wrapper classes efficiency compared to the primitive types. The representation of the primitive types as classes, other things being equal must result in a performance loss.

Create a project that you can call *Wrappers*. Add the following method to the main class:

```
private static void test1()
{
 long t1 = System.nanoTime();
 double sum = 0;
 for (int i = 0; i < T; ++i) sum += Math.sqrt(i);
 long t2 = System.nanoTime();
 System.out.println(sum);
 System.out.println(t2 - t1);
}
```

Here is the method *nanoTime()* a method that reads the hardware clock and returns the time in nano seconds. *T* is a constant that indicates the number of loop iterations to be performed. Test the method by calling it from *main()*. The loop must iterate many times to get a time that you can measure.

Then write a method *test2()* that do exactly the same as *test1()* when the second and third statement should be replaced by:

```
Double sum = 0D;
for (Integer i = 0; i < T; ++i) sum += new Double(Math.sqrt(i));
```

Compare the two methods and the time difference.

The following methods determines the first prime number greater than or equal to a constant N:

```
private static void test3()
{
 long t1 = System.nanoTime();
 long n = N;
 while (!isPrime1(n)) ++n;
 long t2 = System.nanoTime();
 System.out.println(n);
 System.out.println(t2 - t1);
}

public static boolean isPrime1(long n)
{
 if (n == 2 || n == 3 || n == 5 || n == 7) return true;
 if (n < 11 || n % 2 == 0) return false;
 for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2) if (n % t == 0)
 return false;
 return true;
}
```

The idea is that the method takes a long time, if N is large. Test the method for large values of N. Finally, write a test method *test4()*, which is similar to the above, but uses a prime method *isPrime2(Long n)*, ie a method wherein the parameter's type is a wrapper for a *long*. The loop for testing for prime numbers must also use variables of the type *Long*. After writing the method, compare with *test3()*.

The result should be that you observe that it costs something in time to use wrapper objects rather than primitive values, but actually surprisingly little. The reason is that the compiler due to auto boxing knows the wrapper classes, and thus to a large extent can use them as if it were primitive types.

Another way to measure the time is

```
long t1 = Calendar.getInstance().getTimeInMillis();
```

that measure the time in milliseconds. Write a test method that creates an *ArrayList<Integer>* with a million random integers. Test how many milliseconds it takes to sort the list. Remember that you can sort an *ArrayList* as follows:

```
Collections.sort(list);
```

Note that you can not create an ArrayList with elements of the type *int*. The type must be a class type as such an *Integer*.

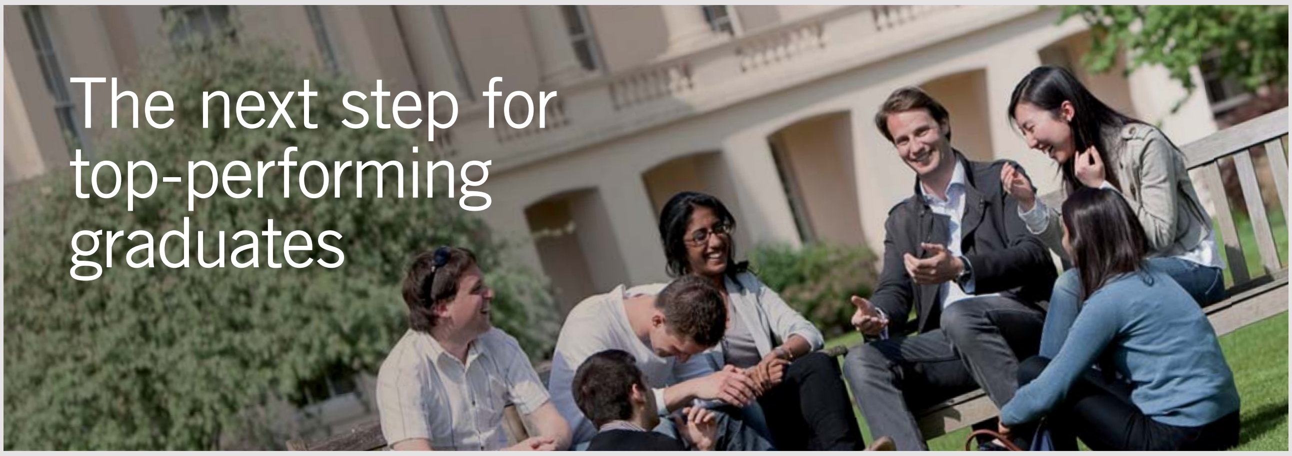
3 STRINGS

The type *String* is a class, although in most cases you uses a *String* as the other simple data types. In most cases, the difference is not of great importance, but in some contexts it is something you should be aware of.

You should be aware that the class has a number of methods to manipulate strings, and it pays to investigate which methods are available. Many of them I have used already and more are used in the course of the books. The class also have useful static methods. I will not discuss these methods here, but the application will appear in the books examples, and I've already used many of these methods.

If you examine the documentation for the *String* class, you will see the definition

```
public final class String
 extends Object implements Serializable, Comparable<String>, CharSequence
```



Masters in Management

Designed for high-achieving graduates across all disciplines, London Business School's Masters in Management provides specific and tangible foundations for a successful career in business.

This 12-month, full-time programme is a business qualification with impact. In 2010, our MiM employment rate was 95% within 3 months of graduation*; the majority of graduates choosing to work in consulting or financial services.

As well as a renowned qualification from a world-class business school, you also gain access to the School's network of more than 34,000 global alumni – a community that offers support and opportunities throughout your career.

London Business School

For more information visit www.london.edu/mm, email mim@london.edu or give us a call on +44 (0)20 7000 7573.

* Figures taken from London Business School's Masters in Management 2010 employment report

This means that the class is defined *final* and thus can not be inherited. Moreover, you can see that the class implements three interfaces. The second says that strings can be ordered and exactly tells the interface, that the class implements a method

```
compareTo(String str)
```

which is the comparison method for strings that compares strings in alphabetical order. When the class *String* implements this interface, it means among other things that strings can be sorted with Java's sorting methods. The last interface tells something about what a string is. Internally is a *String* is an array of characters, and you can refer to the individual characters with the method *charAt()*. An important characteristic of the class *String* is that it is *immutable*, and that means that have you created a *String*, it can not be changed. For instance you can not change a character in a string. If you, for example has the string

```
String s = "abcdefg";
```

and you want to change the character *d* to a big *D*, you must write something like the following:

```
s = s.substring(0, 3) + "D" + s.substring((4));
```

Here, you take a substring consisting of the first three characters and concatenates it with string consisting of the character *D*. This result is then concatenated with the substring consisting of all characters from index 4 to the end of the string. Concatenation of two strings create a new *String* object, and the above statement will create two objects, and the variable *s* is set to refer to the result object instead of the original string. At the class in this way is immutable sounds complicated, and it is at times too, but the reason is performance, where it is important that the creation of string is effective and a string not fills more than necessary. In practice it is not something you think much about when the compiler largely treats strings as other simple types, and as the *String* class has many methods.

However, in special cases, it is important that you are aware that manipulation of strings constantly creates new objects. If you considers the following method (a method in the project *Strings*):

```
private static void test01()
{
 String s = "";
 long t1 = getTime();
 for (int i = 0; i < 100000; ++i) s += "A";
 long t2 = getTime();
 System.out.println(t2 - t1);
}

public static long getTime()
{
 return new GregorianCalendar().getTimeInMillis();
}
```

it creates a string consisting 100,000 occurrences of the character *A*, but the string is built by adding one character a time. The original string *s* is extended by concatenation (the operator +=). This means that the method creates 100,000 new objects, and each time must create a new object on the heap, and the contents of the old object must be copied to the new object. On both sides of the loop, I have read the hardware clock and finally the method prints how long it took to execute the loop. It depends of course of the machine, but on a (not very fast) machine, it has taken 6554 milliseconds and therefore about 6½ seconds, which is a long time.

3.1 STRINGBUILDER

Now, it is of course a bit extremely to build a string on this way, but there are applications where there is a need to perform many operations on strings. To this end, there is a class *StringBuilder*, which is a class where you can manipulate the individual characters in a string, and represents a string that can be expanded without the need to create a new object. In principle, a *StringBuilder* is the same as an *ArrayList*, but simply a list where the elements are of the type *char*. If the task was to build a string as above, you could do it with the following method, which uses a *StringBuilder*:

```
private static void test02()
{
 StringBuilder b = new StringBuilder();
 long t1 = getTime();
 for (int i = 0; i < 100000; ++i) b.append("A");
 String s = b.toString();
 long t2 = getTime();
 System.out.println(t2 - t1);
}
```

If you execute the method on the same machine, it takes 9 milliseconds, and thus there is a very big difference in performance. You should note that the algorithm (method) is the same, and that the improvement therefore solely due to the class *StringBuilder*, and that it is no longer necessary to create and copy the 100000 objects.

In the two methods I have read the hardware clock using the method *getTime()*, which returns the number of milliseconds after 00:00:00:000. The method creates a *GregorianCalendar* object that is defined by the interface *Calendar*. The constructor of the class *GregorianCalendar* initializes the object by reading the hardware clock. The type defines and represents a date and time for a particular day. A *Calendar* has many methods, and here among other things a method *get()* which returns a specific value for a time where a constant indicates the value you want to get. The class *GregorianCalendar* is the standard class in Java for dates and times.

3.2 STRINGTOKENIZER

Java has a class for treatment of strings called *StringTokenizer*. I have used the class previously in the last example in the book Java 3 to split a string into tokens, but it is generally used to split a string in substrings, which are separated by one or more separation characters. As an example, the following method defines a string that consists of integers separated by either a comma or semicolon:

Tuleva DI tai tietojenkäsittelytieteilijä, edunvalvojasi työelämässä on TEK.

TEKin jäsenenä saat myös tietoa, turvaa, neuvontaa ja lukuisia rahanarvoisia etuja.

Opiskelijalle jäsenyys on maksuton.
Lue lisää *www.tek.fi/opiskelijat*

Jos sinulla on yliopistotason tutkinto ja olet jo työelämässä,
lue lisää *www.tek.fi/jasenyys*

Liity nyt!
www.tek.fi/liity

TEK
TEKNIIKAN AKATEEMISET

```
private static void test03()
{
 StringTokenizer tk = new StringTokenizer("2;3,5;7,11;13,17;19", ",;");
 int sum = 0;
 while (tk.hasMoreTokens()) sum += Integer.parseInt(tk.nextToken());
 System.out.println(sum);
}
```

The method uses a *StringTokenizer* to split the string in numbers and determine their sum.

PROBLEM 1

In Java 2, I have created a class library named *PaLib*. In this problem, you must expand this library with a new class.

Start with an copy of the library and open the copy in NetBeans. To the package *palib.util* you must add the following class, which defines some methods that may be useful for manipulating strings:

```
package palib.util;

import java.util.*;
/**
 * Class defining methods of operations on strings.
 */
public abstract class Str
{
 /**
 * Method which cuts off a string of specific length n.
 * If the string length is greater than n, the operation is ignored and the
 * string is returned unchanged.
 * @param s The string that must be cut off
 * @param n The length of the resulting string
 * @return The string cut to length n
 */
 public static String cut(String s, int n)
 {
 …
 }

 /**
 * Method as left adjusts a string in a field of width n.
 * If the string length is greater than n, the operation is ignored and the
 * string is returned unchanged.
 * @param s The string needs that has to be adjusted
 * @param n The width of the field
 * @param c The padding char that field has to be filled with.
```

```
  * @return The string left adjusted in a field of width n.
  */
  public static String left(String s, int n, char c)
  {
  …
  }

  /**
  * Method as right adjusts a string in a field of width n.
  * If the string length is greater than n, the operation is ignored and the
  * string is returned unchanged.
  * @param s The string needs that has to be adjusted
  * @param n The width of the field
  * @param c The padding char that field has to be filled with.
  * @return The string right adjusted in a field of width n.
  */
  public static String right(String s, int n, char c)
  {
  …
  }

  /**
  * Method which centers a string in a field of width n.
  * If the string length is greater than n, the operation is ignored and the
  * string is returned unchanged.
  * @param s The string needs that has to be adjusted
  * @param n The width of the field
  * @param c The padding char that field has to be filled with.
  * @return The string adjusted center in a field of width n.
  */
  public static String center(String s, int n, char c)
  {
  …
  }

  /**
  * Returns the sum of a number of decimal numbers delimited by the character c.
  * If an item (number) can not be parsed into a double, it should just be
  * igonered.
  * @param s String representing a number of numbers separated by the character c
  * @param c The character that separates the numbers
  * @return The sum of the numbers
  */
  public static double sum(String s, char c)
  {
  …
  }
}
```

You should note that the class is defined *abstract*. This means that the class can not be instantiated, and since it has only static methods, it makes nor no sense.

Once you have written the class, you should test your class library with the following test program:

```
package strprogram;

import palib.util.*;

public class StrProgram
{
 public static void main(String[] args)
 {
 String s = "1234567890";
 System.out.println(Str.cut(s, 8));
 System.out.println(Str.left(s, 15, '#'));
 System.out.println(Str.right(s, 15, '#'));
 System.out.println(Str.center(s, 15, '#'));
 System.out.println(Str.sum("2 + 3 + 5.25 + 7.75", '+'));
 }
}
```

#2020Resolutions
To create a digital learning culture
CHECK
bookboonglobal
Unlock your company's full potential with Bookboon Learning. We have the highest staff usage rates in the learning industry.
Find out why ▶▶▶

```
12345678
1234567890#####
#####1234567890
###1234567890##
18.0
```

I have called the test project *StrProgram*.

3.3 REGULAR EXPRESSIONS

A regular expression is a concept that is closely related to strings, and you can use regular expressions to define patterns for strings, and then you can ask if another string has one or more substrings that match this pattern. In Java, regular expressions are implemented by two classes, which are called *Pattern* and *Matcher*. The classes are defined in the package *java.util.regex*. The syntax for regular expressions is relatively complex and the concept is introduced most easily by means of examples. I will therefore start with the following method:

```
private static void test04()
{
  while (true)
  {
   String udt = enter("Regular expression: ");
   if (udt.length() == 0) break;
   Pattern pattern = Pattern.compile(udt);
   while (true)
   {
    String str = enter("Search string: ");
    if (str.length() == 0) break;
    Matcher matcher = pattern.matcher(str);
    boolean found = false;
    while (matcher.find())
    {
     System.out.printf(
      "The text \"%s\" found with start index %d and end index %d\n",
      matcher.group(), matcher.start(), matcher.end());
     found = true;
    }
   if(!found) System.out.println("No match found");
  }
 }
}
```

The method is a simple console application that runs in a dialog with the user. It uses the method *enter()*, which is a simple input method for entering a string. The program runs in an infinite loop. For each repetition, the user must enter a string for a regular expression, and if it is not the empty string the program creates a *Pattern* object that represents the regular expression. This is done by using a static method *compile()* in the class *Pattern*. Next, the program starts an inner loop, which is also an infinite loop. Here the user must enter the string that should match the regular expression, and if it is not the empty string the program creates a *Matcher* object using the regular expression *pattern*. For this *Matcher* object *match* is called a method *find()*, which searches for a substring that matches the regular expression. As long as there is such a substring, it is printed together with its start and end index.

I will use the above method to illustrate the syntax of regular expressions, and the meaning is that you should continue with your own expressions.

The simplest regular expression is simply a string and where a second string that contains the first string is matching the expression. Below is an example of a run of the above program:

```
Regular expression: Knud
Search string: Det er fra Knud og Agnes Knudsen Borremose
The text "Knud" found with start index 11 and end index 15
The text "Knud" found with start index 25 and end index 29
Search string:
```

That is, I as regular expression entered

```
Knud
```

while I, as a search string entered

```
Det er fra Knud og Agnes Knudsen Borremose
```

The result shows that the search string contains two substrings that matches the regular expression.

To specify patterns you needs special characters, called meta-characters that has a special meaning in a regular expression. There are following characters:

```
< ( [ { \ ^ - = $ ! | ] } ) ? * + . >
```

If there is a need for these characters not to be interpreted, but is perceived as common characters in the text, you can prefix the character a backslash.

One of the basic patterns are character classes that you define as follows:

-	`[abc]`	all characters *a*, *b* and *c*
-	`[^abc]`	all characters that is not *a*, *b* and *c* (also called negation)
-	`[a-zA-Z]`	all characters from *a* to *z* and from *A* to *Z* (union)
-	`[a-d[m-p]]`	all characters from *a* to *d* and *m* to *p* (same as [a-dm-p])
-	`[a-z&&[def]]`	all characters *d*, *e* and *f* (intersection)
-	`[a-z&&[^bc]]`	all characters from *a* to *z* but not *b* and *c* (set difference)
-	`[a-z&&[^m-p]]`	all characters from *a* to *z* but not *m* to *p* (same as [a-lq-z])

If you, as an example, consider the regurlar expression

```
[hkr]at
```

WORK WITH US
to gether ness
Reduce Reuse Recycle
It's only an opportunity if you act on it
IKEA.SE/STUDENT
Save water. Shower together
everyone deserves good design
IKEA®
© Inter IKEA Systems B.V. 2009

it matches the substrings *hat*, *kat* and *rat*:

```
Regular expression: [hkr]at
Search string: hat
The text "hat" found with start index 0 and end index 3
Search string: kat
The text "kat" found with start index 0 and end index 3
Search string: rat
The text "rat" found with start index 0 and end index 3
Search string: vat
No match found
Search string: it is both hat and kat but not vat
The text "hat" found with start index 11 and end index 14
The text "kat" found with start index 19 and end index 22
Search string:
```

As another example, the pattern

```
num[^123]
```

matches all substrings, which consists of the word *num* followed by a character which is not 1, 2 or 3:

```
Regular expression: num[^123]
Search string: num
No match found
Search string: num1
No match found
Search string: num4
The text "num4" found with start index 0 and end index 4
Search string:
```

The regular expression

```
[A-G]
```

matches all uppercase letters from A to G:

```
Regular expression: [A-G]
Search string: Anders Andersen
The text "A" found with start index 0 and end index 1
The text "A" found with start index 7 and end index 8
Search string: Harald Hen
No match found
Search string: Harald Gormssøn
The text "G" found with start index 7 and end index 8
Search string:
```

The pattern

```
[^0-9]
```

matches any character that is not a digit:

```
Regular expression: [^0-9]
Search string: abc
The text "a" found with start index 0 and end index 1
The text "b" found with start index 1 and end index 2
The text "c" found with start index 2 and end index 3
Search string: 0x123
The text "x" found with start index 1 and end index 2
Search string: 1234
No match found
Search string:
```

The following expression is an example of a union that matches all digits and lowercase letters between a and f:

```
Regular expression: [0-9[a-f]]
Search string: xya1z
The text "a" found with start index 2 and end index 3
The text "1" found with start index 3 and end index 4
Search string: xyz
No match found
Search string:
```

Below is an example of an intersection. The goal is to show the syntax, for the regular expression could instead be writen as [h-n].

```
Regular expression: [a-n&&[h-u]]
Search string: Preben
The text "n" found with start index 5 and end index 6
Search string: Frede
No match found
Search string:
```

Some specific character classes can be identified by a symbol:

- .   Any character
- \d   The 10 digits
- \D   All characters that not are a digit

- \s A *whitespace*: [\t\n\f\r]
- \S All characters that not are a whitespace
- \w Lettes and digits: [a-zA-Z0-9]
- \W All other characters: [^\w]

Example of a regular expression, which is a dot:

```
Regular expression: .
Search string: 123
The text "1" found with start index 0 and end index 1
The text "2" found with start index 1 and end index 2
The text "3" found with start index 2 and end index 3
Search string:
```

Example of a regular expression, which matches a digit:

```
Regular expression: \d
Search string: ab12cd
The text "1" found with start index 2 and end index 3
The text "2" found with start index 3 and end index 4
Search string:
```

ericsson.
com

Shaping tomorrow's world – today
Our business is at the heart of a connected world – a world where communication is empowering people, business and society. Our networks, telecom services and multimedia solutions are shaping tomorrow. And this might just be your chance to shape your own future.

It's a people thing
We are looking for high-caliber people who can see the opportunities, people who can bring knowledge, energy and vision to our organization. In return we offer the chance to work with cutting-edge technology, personal and professional development, and the opportunity to make a difference in a truly global company.

We are currently recruiting both new graduates and experienced professionals in four areas: **Software, Hardware, Systems** and **Integration & Verification.**

Are you ready to shape **your** future? Begin by exploring a career with Ericsson. Visit www.ericsson.com/join-ericsson

ERICSSON

Example of a regular expression, which is a whitespace:

```
Regular expression: \s
Search string: 1 2 3
The text " " found with start index 1 and end index 2
The text " " found with start index 3 and end index 4
Search string:
```

There are also some options to specify that a pattern must occur several times and for each there are even three variants. If X represents a pattern the syntax is

X?	X??	X?+	X occurs once or not at all
X*	X*?	X*+	X occurs several times or possibly not at all
X+	X+?	X++	X occurs at least once
X{n}	X{n}?	X{n}+	X occurs exactly n time
X{n,}	X{n,}?	X{n,}+	X occurs at least n time
X{n,m}	X{n,m}?	X{n,m}+	X occurs at least n times and at most m times

In principle, these operators are simple enough, but it is not always so easy to predict the outcome:

```
Regular expression: a?
Search string: 123
The text "" found with start index 0 and end index 0
The text "" found with start index 1 and end index 1
The text "" found with start index 2 and end index 2
The text "" found with start index 3 and end index 3
Search string:
```

a? matches a substring consisting of zero or one character *a*. If *x* denotes this pattern, one can perceive the search string 123 as *x1x2x3x*, and you will therefore have 4 matches. Therefore, the following pattern results in 6 matches:

```
Regular expression: a?
Search string: 1a2a3
The text "" found with start index 0 and end index 0
The text "a" found with start index 1 and end index 2
The text "" found with start index 2 and end index 2
The text "a" found with start index 3 and end index 4
The text "" found with start index 4 and end index 4
The text "" found with start index 5 and end index 5
Search string:
```

a* matches any number of the character *a*, so you end with 5 matches:

```
Regulært udtryk: a*
Søgestreng: 12aaaaaaaaaaaa3
Teksten "" fundet med start på indeks 0 og slut på indeks 0
Teksten "" fundet med start på indeks 1 og slut på indeks 1
Teksten "aaaaaaaaaaaa" fundet med start på indeks 2 og slut på indeks 14
Teksten "" fundet med start på indeks 14 og slut på indeks 14
Teksten "" fundet med start på indeks 15 og slut på indeks 15
```

As a+ requires at least one character *a*, you below get respectively no and 1 match:

```
Regular expression: a+
Search string: 123
No match found
Search string: 123aaa45
The text "aaa" found with start index 3 and end index 6
Search string:
```

The above examples show how the method *find()* in the class *Matcher* works. It searches over again in the search string until it finds a substring that matches the regular expression. If it finds a match, the index has reached the first character after the string. The next search will start from that location. Consider again a search similar to the above:

```
Regular expression: a?
Search string: aaa
The text "a" found with start index 0 and end index 1
The text "a" found with start index 1 and end index 2
The text "a" found with start index 2 and end index 3
The text "" found with start index 3 and end index 3
Search string:
```

1. The search starts from the begining, and the index is 0. The search ends when the index is 1 when the substring "a" matches the regular expression.
2. Next search stops with the index in place 2, where they have found the next substring "a" that matches.
3. The third search finds the substring "a" and stop index of 3.
4. Finally stops the last search when the end of the string is reached, and at that time the index is still 3 and you have found the empty string that matches the regular expression.

Below is an example that shows a bit of the same, but the first search stops first when the index is 3:

```
Regular expression: a*
Search string: aaa
The text "aaa" found with start index 0 and end index 3
The text "" found with start index 3 and end index 3
Search string:
```

This also applies to the example below, but the last search, which results in the empty string does not match the regular expression:

```
Regular expression: a+
Search string: aaa
The text "aaa" found with start index 0 and end index 3
Search string:
```

The above examples show that you often get a different result than expected. If, for example you want an expression that matches certain number of characters, the syntax often will be something like the following:

Löydä koulutuksesi!
Studentum.fi auttaa sinua löytämään itsellesi sopivan opiskelupaikan koulutusviidakosta. Etsi, vertaile ja löydä oma koulutuksesi!
Studentum.fi
Löydä koulutuksesi!

```
Regular expression: a{3}
Search string: aaaaa
The text "aaa" found with start index 0 and end index 3
Search string:
```

If you want to search a group of characters used parentheses, where the following expression matches sekvenses 123123:

```
Regular expression: (123){2}
Search string: 891231234512367
The text "123123" found with start index 2 and end index 8
Search string:
```

As another example, the following pattern matches 3 of the characters *a*, *b* and *c*:

```
Regular expression: [abc]{3}
Search string: abdbbbeabccbaf
The text "bbb" found with start index 3 and end index 6
The text "abc" found with start index 7 and end index 10
The text "cba" found with start index 10 and end index 13
Search string:
```

As shown in the table above there are three options for specifying quantifiers. As mentioned the search in the search string moves the index until a match is found. If the quantifiers in the first column are used the index is moved as far as possible. A dot means the character class consisting of all characters, and the pattern

```
.*abc
```

therefore means 0 or more characters followed by abc. As the index moved as far to the right as possible the result of the following search is only in one match:

```
Regular expression: .*abc
Search string: xabcxxxxxabc
The text "xabcxxxxxabc" found with start index 0 and end index 12
Search string:
```

If the quantifiers in the second column are used, the search stops as soon as a match is found. Therefore results the following search two matches:

```
Regular expression: .*?abc
Search string: xabcxxxxxabc
The text "xabc" found with start index 0 and end index 4
The text "xxxxxabc" found with start index 4 and end index 12
Search string:
```

Finally if the quantifiers in the third column are used, the index is moved as far as possible. Because .* matches all characters, the index is moved to the end of the string – and past abc. Therefore, the following search has no match:

```
Regular expression: .*+abc
Search string: xabcxxxxxabc
No match found
Search string:
```

There are also some options to specify where in the search string a match must occur:

- ^	the start of the line
- $	the end of the line
- \b	the start of a word
- \B	not in the start of a word
- \A	the start of the search string
- \G	the start of the previous match
- \z	the start of the search string

Correspondingly, the following searches results in a single match:

```
Regular expression: ^abc
Search string: abcabcabc
The text "abc" found with start index 0 and end index 3
Search string:
Regular expression: abc$
Search string: abcabcabc
The text "abc" found with start index 6 and end index 9
Search string:
```

The above examples demonstrate the basics regarding regular expressions, but there are several options, and the classes *Pattern* and *Matcher* also has other useful methods. Here I refer to the documentation, and will instead end this introduction to regular expressions with a few examples.

In Java, the name of a variable start with a letter, and then there should follow any number of characters consisting of letters, digits and the character _. In addition, it is recommended that a variable name always starts with a lowercase letter. If one decides that it's the rules, a variable is defined by using the following regular expressions:

```
private static void test05()
{
  Pattern pattern = Pattern.compile("^[a-z]+[a-zA-Z_0-9]*$");
  for (String str = enter("? "); str.length() > 0; str = enter("? "))
  {
   Matcher matcher = pattern.matcher(str);
   System.out.println(matcher.find());
 }
}
```

If you executes the method, you can enter strings, and the method will validate where the string is a valid variable. The pattern says that the string must start with just a small letter:

```
[a-z]+
```

Next, the strings must end with any number of characters that are letters, digits or _:

```
[a-zA-Z_0-9]*$
```

e-learning for kids
●The number 1 MOOC for Primary Education
●Free Digital Learning for Children 5-12
●15 Million Children Reached

About e-Learning for Kids Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.

Similarly, the following method validates whether a string represents a binary number (a string consisting of 0 and 1):

```
private static void test06()
{
 Pattern pattern = Pattern.compile("^[01][01]*$");
 for (String str = enter("? "); str.length() > 0; str = enter("? "))
 {
  Matcher matcher = pattern.matcher(str);
  System.out.println(matcher.find());
 }
}
```

As a complex example is shown a method using a regular expression to validate an email address. Note that I have already used the method in the book Java 3.

```
public static boolean isMail(String mail)
{
 return Pattern.compile(
  "^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.
  [0-9]{1,3}\\.[0-9]{1,3}\\])|(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,}))$").
 matcher(mail).matches();
}
```

It is a very complex expressions, and also it uses rules and operators that I not have highlighted above, so I will try to explain the expression. It starts with a character class:

```
[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+
```

It defines the small and capital letters, digits and a number of other special characters. You should note that when a special character is defined in a character class, it is unnecessary to escape them with a backslash. The exception is [,] and – because these characters are used to define the class. An email address consists of two parts separated by the @ character, as we call respectively the first name and the last name. The above character class defines thus what characters the first name may consist of, and that there must be at least one of these characters. You should note that the class allows many other characters than what is usual in mail addresses, but they are actually legal – at least if you includes mail addresses from all countries.

After the above character class follows the separator @, and then the last name that is made up of two terms, so that a string should match the one or the other. You define this selection with the character |, and the first expression is therefore:

```
(\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\])
```

The expression matches a bracket begin, four integer of at least 1 and no more than 3 digits separated of a dot, and last a final bracket. This means that the expression matches an IP address in brackets.

The second expression has the form:

```
(([a-zA-Z\\-0-9]+\\.)+[a-zA-Z]{2,})
```

Here is

```
[a-zA-Z\\-0-9]+
```

a character class that matches uppercase and lowercase letters, a hyphen and digits, and there must be at least one of these characters. Next, follow a dot. Of these groups must be at least one:

```
([a-zA-Z\\-0-9]+\\.)+
```

Finally there must be at least two letters. The result is that the last name is either an IP address or a server name. The entire expression is surrounded by the characters ^ and $, which means that there must be no front or behind the expression.

EXERCISE 2

In this exercise, you should extends the class library *PaLib*. Add the method *isMail()* to the class *Str*. You must also add the following methods when they should be implemented using a regular expression:

```
/**
 * Validates where a string is hexadecimal integer without sign, when the syntax
 * should be that of Java, where the numbers starts with 0x.
 * @param str The string to be validated
 * @return true, if str represents hexadecimal integer without sign
 */
public static boolean isHex(String str) { … }

/**
 * Validates whether a string is a long when the number must either be 0, or an
 * integer which may have a sign and must not start with 0, but otherwise
 * have maximum 17 digits.
 * @param str The string to be validated
 * @return true, if str represents a long
 */
public static boolean isLong(String str) { … }
```

```
/**
 * Validates where a string is a double when the number may start with a sign and
 * there must be a decimal point followed by at least one digit. Moreover, it
 * should be possible to end the number with a eksponent part.
 * @param str The string to be validated
 * @return true, if str represents a double
 */
public static boolean isDouble(String str) { … }
```

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

I joined MITAS because
I wanted **real responsibility**
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

4 INNER CLASSES

In Java, you can use nested classes that are classes within classes, and you can in fact can even define a class inside a method. In most cases defines a class a thing or a concept concerning the program's problem area, and all that is said about classes into this place still apply, but sometimes you need small classes only to be used internally in a another class, and in such cases you can define an inner class. Seen from the finished program it plays no role, and the program is no more or less effective for that, but you get a better encapsulation, which in turn can result in code that is easier to read and understand.

It sounds simple to define inner classes, and it also is, but there is nevertheless a lot of details that you needs to know, and that's what this chapter covers. Basically, you can define classes as follows:

```
class OuterClass
{
   …
   static class NestedClass
   {
    …
   }
   class InnerClass
   {
    …
  }
}
```

where the class *OuterClass* has two inner classes. The difference is that one is *static*, while the other is not. A static inner classes are usually called a *nested* class, while a non static is called an *inner* class. An *inner* class can refer to all in the outer class regardless of whether it is private or not. A nested, in turn, can not refer to the instance variables or non-static methods in the outer class, unless it has an object. Similar to other static members in a class, references to a nested class must be done via the name of the outer class, for example something like the following:

```
OuterClass.NestedClass obj = new OuterClass.NestedClass();
```

An inner class is in the same way as other members of the outer class associated with an object of the outer class, and this means, that an inner class can not have static members. Stated slightly differently, objects that are instances of an inner class, only exists in an instance of the outer class. If *outer* is an object of type *OuterClass*, you can create an object of the type *InnerClass* as follows:

```
OuterClass.InnerClass obj = outer.new InnerClass();
```

Consider the following class *OuterClass*:

```
class OuterClass
{
 public int x = 0;

 class InnerClass
  {
  public int x = 1;

  void print(int x)
  {
   System.out.println("x = " + x);
   System.out.println("this.x = " + this.x);
   System.out.println("OuterClass.this.x = " + OuterClass.this.x);
  }
 }
}
```

that defines an inner class called *InnerClass*. Both *OuterClass* and *InnerClass* has a variable called *x*, and because the two variables has the same name they can shade for each other. The class *InnerClass* has a method called *print()*, and it has a parameter, that is also called *x*. When the method refers to *x* (the first statement), it is the parameter that is referred to. When writing *this.x* (the second statement) it is as mentioned earlier the instance variable that is referenced, and that is the instance variable in the class *InnerClass*. If you writes *OuterClass.this.x* (the third statement) is the instance variable in the class *OuterClass* that it is refered. Performing the following method:

```
private static void test()
{
 OuterClass outer = new OuterClass();
 OuterClass.InnerClass inner = outer.new InnerClass();
 inner.print(23);
}
```

you get the result:

```
x = 23
this.x = 1
OuterClass.this.x = 0
```

The same convention for solution of problems with the same names are used if there instead of a variable is a method.

4.1 ITERATORS

If you have an ArrayList:

```
ArrayList<Integer> list = new ArrayList();
```

you can iterate over the elements in the list as follows:

```
for (Integer t : list) System.out.println(t);
```

FACTCARDS

Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?

Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.

The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.

VISIT FACTCARDS.NL

I have previously mentioned the iterator pattern, and the above statement is legal because the class *ArrayList* implements this pattern, and thus the class implements the interface *Iterable<Integer>*, which defines a principle of how to traverse a collection. In fact, the above statement is translated to something like the following:

```
for (Iterator<Integer> itr = list.iterator(); itr.hasNext(); )
 System.out.println(itr.next());
```

which is an ordinary *for* statement. I would later treat the iterator pattern in details and including how it is implemented, but as an example of the use of inner classes I will below shows a simple class that represents the primes:

```
package primeprogram;

import java.util.*;

public class Primes implements Iterable<Long>
{
 private long p = 2;
 private long m;

 public Primes(long m)
 {
  this.m = m;
 }

 public Iterator<Long> iterator()
 {
  p = 2;
  return new PrimeIterator();
 }

 private class PrimeIterator implements Iterator<Long>
 {
  public boolean hasNext()
   {
  return p <= m;
   }
 public Long next()
 {
   long t = p;
   if (p == 2) p = 3; else for (p += 2; !isPrime(p); p += 2);
   return t;
  }
 }

 public static boolean isPrime(long n)
 {
  if (n == 2 || n == 3 || n == 5 || n == 7) return true;
  if (n < 11 || n % 2 == 0) return false;
  for (long t = 3, m = (int)Math.sqrt(n) + 1; t <= m; t += 2) if (n % t == 0)
  return false;
  return true;
 }
}
```

When you create an instance of the class, you specify a positive integer, and the object represent all primes less than or equal to this number. The class makes it possible to iterate the prime numbers that it represents. That is you can write something like the following:

```
Primes primes = new Primes(100);
for (long p : primes) System.out.print(p + " ");
System.out.println();
```

The class implements the interface *Iterable<Long>*, which means that the class must have a method called *iterator()*, which returns an object of the type *Iterator<Long>*. An iterator is an object that can refer to a particular item in a collection (and here in *Primes*), and has two methods where *hasNext()* tests whether you have reached the end while the *next()* returns the item that the iterator indicate and at the same time moves the iterator forward. *Iterator<Long>* is an interface, and you has to write a class that implements this interface. This class will be used to instantiate an object that is an internal property of the class *Primes* and the class is written as a inner class called *PrimeIterator*. Note that the class is private so it is only known inside the class *Primes*.

The class *Primes* have two instance variables, the first *p* is the prime, as currently is reached, while the other *m* is the upper limit of the primes, the class should represent. When *PrimeIterator* is an inner class, it can refer to these variables, and the implementation of the class is quite trivial. The class *Primes* implements the method *iterator()* – defined by the interface *Iterable<Long>* – and it sets the start of the primes to 2 and returns an object of the type *PrimeIterator*.

The most important class is the inner *PrimeIterator* and as a typical application of an inner class, but it is also an example of how to implement the iterator pattern with a custom iterator.

As can be seen from the above example, an inner class can be *private*, and can also be both *public* and *protected*, and the meaning is the same as that for other members of a class.

It is also possible to define a class in a method:

```
private static void test()
{
  int[] arr = { 2, 3, 5, 7, 11, 13, 17, 19 };

  class Counter
   {
    private int n = 0;

    public int value()
    {
     int t = n;
     n = (n + 1) % arr.length;
     return t;
    }
```

WHILE YOU WERE SLEEPING...
www.fuqua.duke.edu/whileyouweresleeping
DUKE
THE FUQUA SCHOOL OF BUSINESS

```
  }

  Counter c = new Counter();
  for (int i = 0; i < 10; ++i) System.out.print(arr[c.value()] + " ");
  System.out.println();
}
```

Such a class is called a *local class* and *Counter* above is an example. In this case there is no good reason for the local class, but you should note that the local class can refer to local variables in the method by which it is a part. You will also notice that the class must be defined before it can be instantiated.

Consider then the following class that defines an iterator which iterates over the factorials:

```
package factorials;

import java.util.*;

public class Factorial implements Iterable<Long>
{
  public Iterator<Long> iterator()
  {
   return new Iterator<Long>()
   {
    private long t = 1;
    private int i = 0;

    public boolean hasNext()
    {
     return i < 20;
    }

    public Long next()
    {
     return t *= ++i;
    }
   };
  }
}
```

The class is called *Factorial* and implements the interface *Iterable<Long>*. It must, therefore, in the same manner as the class *Primes* implement the method *iterator()*, and in this case, the class has no other members. The method *iterator()* must return an object of the type *Iterator<Long>* and hence I has to writte a class that implements this interface. The class does not have to be known outside the method *iterator()*, and can therefore be written as an anonymous class, namely a class which has no name. You should note the syntax:

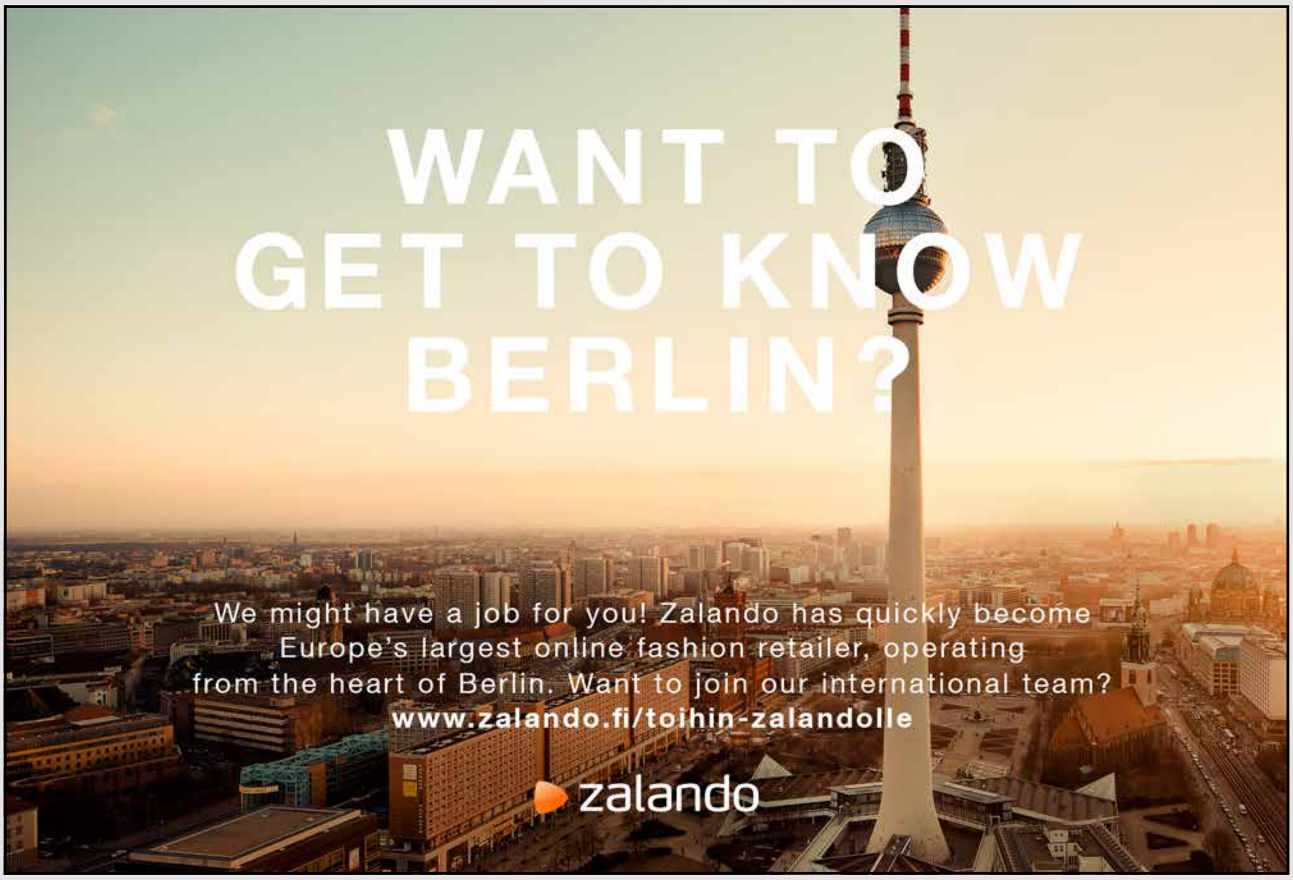
```
return new Iterator<Long>()
{
}
```

and an anonymous class must be defined on the basis of an interface or a class that it inherits. With the class *Factorial* defined you can write a statement like:

```
for (long t : new Factorial()) System.out.println(t);
```

There are many examples of the use of anonymous classes, for example the class *Primes*, that could be written as follows, wherein the iterator is now implemented as an anonymous class:

```
package primeprogram;

import java.util.*;

public class Primes implements Iterable<Long>
{
 private long m;

 public Primes(long m)
 {
  this.m = m;
 }
 public Iterator<Long> iterator()
 {
  return new Iterator<Long>()
  {
   long p = 2;

   public boolean hasNext()
   {
    return p <= m;
   }
   public Long next()
   {
    long t = p;
    if (p == 2) p = 3; else for (p += 2; !isPrime(p); p += 2);
    return t;
   }
  };
 }

 public static boolean isPrime(long n)
 {
  if (n == 2 || n == 3 || n == 5 || n == 7) return true;
  if (n < 11 || n % 2 == 0) return false;
  for (long t = 3, m = (int)Math.sqrt(n) + 1; t <= m; t += 2) if (n % t == 0)
   return false;
  return true;
 }
}
```


WANT TO
GET TO KNOW
BERLIN?
We might have a job for you! Zalando has quickly become
Europe's largest online fashion retailer, operating
from the heart of Berlin. Want to join our international team?
www.zalando.fi/toihin-zalandolle
zalando

EXERCISE 3

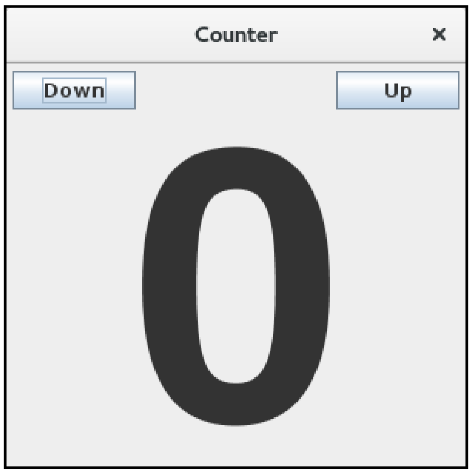
Write a program similar to the program *Factorials* when the program this time should implements a class *Fibonacci*, where you can iterate through the fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, …

The class *Fibonacci* must implement the interface *Iterable<Long>* and should only have a single method that returns the iterator, and the main program should only print the fibonacci numbers. The iterator must as in the class *Factorial* be implemented as an anonymous class. Note that the largest fibonacci number that you can represent as a *long* is the number with index 89.

EXERCISE 4

Write a program called *CounterProgram*, that opens the follewing window:



The program must have a very simple class named *Counter*. The class should have an instance variable of the type *int*, that from start is 0. The class must have three methods:

- *up()*, that increments the counter by 1
- *down()*, that decrements the counter by 1
- *getValue()*, that returns the value of the counter

Center in the window is a label, that shows the value of the counter. When the user click on the button *Up*, the counter should be increased, and when the user click on *Down* the counter should be decreased. The event handler to one of the two buttons must be written as an inner class, and the event handler to the other button must be written as an anonymous class.

4.2 EXAMPLE: ZIPCODES

I will show an example of a program where you can search the Danish ZIP codes. The user can enter a value for the ZIP code and a value for the city name, and the program will then find the elements that matches these values, when the zipcode code must start with the entered value, while the city name must contain the entered value. When comparing the city names, it should not be case-sensitive. I start with a simple model class for a zip code:

```
package postprogram;

public class Post
{
 private String code; // the zip code
 private String city; // name og the town

 public Post(String code, String city)
 {
 this.code = code;
 this.city = city;
 }

 public String getCode()
 {
 return code;
 }

 public String getCity()
 {
 return city;
 }

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

Next there is written a class *Postcodes*, that is an encapsulation of an *ArrayList<Post>*. There are two challenges associated with this class:

1. the arraylist must be initialized with *Post* objects
2. the class should have two iterators to iterates the list sorted respectively by zip code or city name

The first problem is solved by placing a table directly in the code. It is obviously not a very flexible solution, but postcodes does not change frequently, so in this case the solution can be accepted. In the program, post codes are laid out as an array in a nested class. The class is also public, so others can refer to it. Below is a part of the class:

```
public static class Data
{
 private final String[][] table = {
  { "1000", "København K" },
  { "1001", "København K" },
  { "1002", "København K" },
  ….
  { "9982", "Ålbæk" },
  { "9990", "Skagen" }
  };

  public int length()
  {
   return table.length;
  }
```

SIMPLY CLEVER

ŠKODA

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

```
   public String getCode(int n)
   {
    return table[n][0];
   }

   public String getCity(int n)
   {
    return table[n][1];
  }
}
```

The constructor of the class Postcodes uses the class *Data* to initialize an *ArrayList<Post>* with *Post* objects.

The second of the above problems are solved by sorting the *ArrayList*. To sort an array list and for that matter any other list of objects, you should be able to compare the objects and to determine whether an object is greater than another object. It can be solved in several ways, and one way is to specify a comparator, which is an object whose class implements (in this case) the interface *Comparator<Post>*. It defines a single method

```
public int compare(Post p1, Post p2)
{
}
```

which must return -1 if *p1* is less than *p2*, 1 if the *p1* is greater than *p2* and 0 if they are the same. The class is now written, so it has two iterators that starts to sort the post codes by respectively zipcode and city name and the sort order is defined by two inner classes:

```
package postprogram;

import java.util.*;

public class Postcodes implements Iterable<Post>
{
  private ArrayList<Post> list = new ArrayList();

  public Postcodes()
  {
   Data data = new Data();
   for (int i = 0; i < data.length(); ++i)
    list.add(new Post(data.getCode(i), data.getCity(i)));
  }
```

```
 public Iterator<Post> iterator()
 {
  list.sort(new CodeCompare());
  return list.iterator();
 }

 public Iterator<Post> iterator2()
 {
  list.sort(new CityCompare());
  return list.iterator();
 }

 class CodeCompare implements Comparator<Post>
 {
 public int compare(Post p1, Post p2)
  {
   return p1.getCode().compareTo(p2.getCode());
  }
 }

 class CityCompare implements Comparator<Post>
 {
  public int compare(Post p1, Post p2)
  {
   return p1.getCity().compareTo(p2.getCity());
  }
 }

 public static class Data
 {
  ….
 }
}
```

Note first that the class implements the iterator pattern in the usual way. Next, note the two inner classes, each of which defines a comparator. In this case, it is easy to define the comparison, since in either case you should compare strings, and the class *String* supports directly comparison. If you consider the implementation of the method *iterator()*, so it starts to sort the *ArrayList* after zip code, which is done using a method *sort()* and a comparator object. The class has a different method, which also returns an iterator. It's called *iterator2()* and works in exactly the same way, just it sorts with a second comparator.

The above classes can be seen as the model for the program, a model which makes the data available which the program must use. In this case, the program should select some of the objects from the model based on a given search criteria for respectively the zip code and the city name. To this end, I have written a *Controller* class:

```
public class Controller
{
 private Postcodes model = new Postcodes();

 public ArrayList<Post> seek(String code, String city, boolean zipcode)
 {
  ArrayList<Post> list = new ArrayList();
  city = city.toLowerCase();
  if (zipcode)
  {
   for (Post p : model)
    if (p.getCode().startsWith(code) &&
     p.getCity().toLowerCase().contains(city)) list.add(p);
 }
 else
 {
```

Turning a challenge into a learning curve.
Just another day at the office for a high performer.

Accenture Boot Camp – your toughest test yet

Choose Accenture for a career where the variety of opportunities and challenges allows you to make a difference every day. A place where you can develop your potential and grow professionally, working alongside talented colleagues. The only place where you can learn from our unrivalled experience, while helping our global clients achieve high performance. If this is your idea of a typical working day, then Accenture is the place to be.

It all starts at Boot Camp. It's 48 hours that will stimulate your mind and enhance your career prospects. You'll spend time with other students, top Accenture Consultants and special guests. An inspirational two days packed with intellectual challenges and activities designed to let you discover what it really means to be a high performer in business. We can't tell you everything about Boot Camp, but expect a fast-paced, exhilarating and intense learning experience. It could be your toughest test yet, which is exactly what will make it your biggest opportunity.

Find out more and apply online.

Visit accenture.com/bootcamp

• Consulting • Technology • Outsourcing

accenture
High performance. Delivered.

```
  for (Iterator<Post> itr = model.iterator2(); itr.hasNext();)
 {
   Post p = itr.next();
   if (p.getCode().startsWith(code) &&
    p.getCity().toLowerCase().contains(city)) list.add(p);
   }
  }
  return list;
 }
}
```

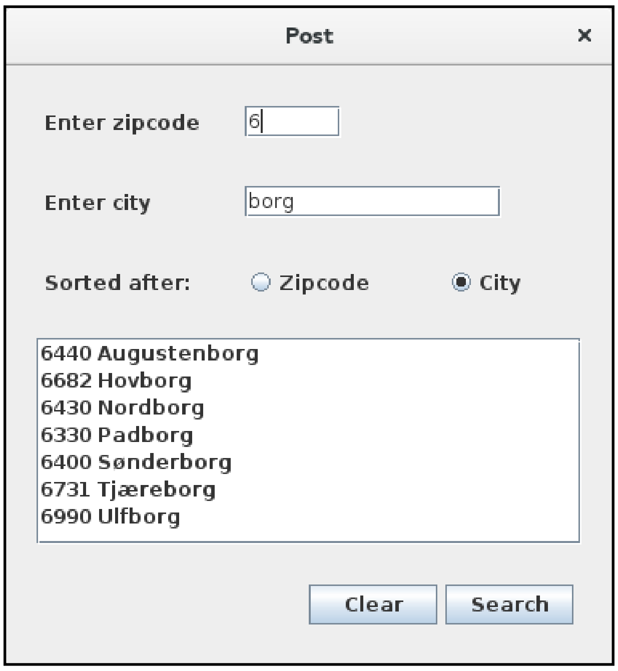
The class creates the model, but otherwise there is only one method to searching. The method has three parameters, where the first two is the search text, respectively for the zip code and the city name. The last parameter specifies whether the results must be sorted by zip code or city name. If the result should be sorted by zip code the standard iterator is used by iterating through the model with the usual syntax:

```
for (Post p : model)
```

Should the result be sorted by city name, *iterator2()* is used and this time you can not use the expanded *for* statement, but must directly refer to the iterator:

```
for (Iterator<Postnummer> itr = model.iterator2(); itr.hasNext();)
```

Now there's just to give the program a user interface and run the program, you get the following window, where there has been a search:



Enter zipcode: 6
Enter city: borg
Sorted after: Zipcode [✓ City]

```
6440 Augustenborg
6682 Hovborg
6430 Nordborg
6330 Padborg
6400 Sønderborg
6731 Tjæreborg
6990 Ulfborg
```

5 ENUMERATIONS

As shown, you can define constants as variables that are *static final*. An alternative is to use a so-called enumeration, and an example might be:

```
package enumerations;
public enum
 WeekDays { MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY, SUNDAY }
```

Weekdays is a reference type that defines seven values and as an example you can write the following method:

```
private static void test01()
{
  String[] days =
  { "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY", "SATURDAY", "SUNDAY" };
  try
  {
   System.out.println(weekday(rand.nextInt(7) + 1));
   System.out.println(weekday(days[rand.nextInt(7)]));
   for (WeekDays day : WeekDays.values()) System.out.println(day);
   System.out.println(WeekDays.WEDNESDAY.compareTo(WeekDays.MONDAY));
   System.out.println(WeekDays.WEDNESDAY.compareTo(WeekDays.FRIDAY));
   System.out.println(WeekDays.WEDNESDAY.compareTo(WeekDays.WEDNESDAY));
  }
  catch (Exception ex)
  {
   System.out.println(ex.getMessage());
  }
}

private static WeekDays weekday(int n) throws Exception
{
 switch (n)
 {
  case 1: return WeekDays.MONDAY;
  case 2: return WeekDays.TUESDAY;
  case 3: return WeekDays.WEDNESDAY;
  case 4: return WeekDays.THURSDAY;
  case 5: return WeekDays.FRIDAY;
  case 6: return WeekDays.SATURDAY;
  case 7: return WeekDays.SUNDAY;
 }
 throw new Exception("Illegal day");
}
```

```
private static WeekDays weekday(String s) throws Exception
{
 return WeekDays.valueOf(s);
}
```

The example should only show the syntax. The first *weekday()* method has an *int* as a parameter and converts it to a value of type *Weekdays*. You should note, how to refer to the individual values with the dot operator, and you should note that the methood's type is *Weekdays*. The second *weekday()* method, has a string as a parameter and returns a *Weekdays* value that matches this string. The two methods are used by the test method. Here you should specifically note that it is possible to loop over an enumeration. You should also note that an enum is comparable and that the order is determined by the order of the values.

Enumerations can help to improve the readability of programs, and it is recommended to use enumerations if a program needs a finite number of values, which are values that do not change. Good examples are week days, month names, colors of playing cards, etc.

Brain power
Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge
SKF

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations.

Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

In Java an *enum* is a little more than just a list of constant values and actually an enum is a class. Consider as an example the following enum (where I have not shown all values):

```
package enumerations;

public enum Kings
{
 GOR (936, 958, "Gorm den Gamle"),
 HA1 (958, 987, "Harald Blåtand"),
 SV1 (987, 1014, "Svend Tveskæg"),
 …
 MA2 (1972, 9999, "Margrethem 2."),
 NON (0, 0, "Interregnum");

 private final int from;
 private final int to;
 private final String name;

 private Kings(int from, int to, String name)
 {
  this.from = from;
  this.to = to;
  this.name = name;
 }

 public int getFrom()
 {
  return from;
 }

 public int getTo()
 {
  return to;
 }

 public String getName()
 {
  return name;
 }

 public String toString()
 {
  return "[" + name() + "] " + name;
 }
 public static Kings getKing(int year)
 {
  for (Kings k : Kings.values()) if (k.from <= year && k.to >= year) return k;
  return Kings.NON;
 }

 public static ArrayList<Kings> getKings(String name)
 {
  name = name.toLowerCase();
  ArrayList<Kings> list = new ArrayList();
  for (Kings k : Kings.values()) if (k.name.toLowerCase().contains(name))
   list.add(k);
  return list;
 }
}
```

The example should show that it is possible to assign arguments or attributes to the individual values. Each value is this time a name of three characters, which is a key for a Danish king. For each value is associated three arguments respectively start and end year of the king's reign and the king's name (or the queen's name). The argument is written after the value, and there must be a corresponding *private* constructor that initializes the constants with the current arguments. In addition the type may have methods, and in this case there are *get* methods to the three arguments, a *toString()* method, a method which can return the reign for a specific year, as well as a search method. Below is shown a test method using the type *Kings*

```
private static void test02()
{
 for (Konge konge : Konge.values()) print(konge);
 System.out.println("------------------------------------------------");
 print(Konge.getKonge(1200));
 System.out.println("------------------------------------------------");
 for (Konge konge : Konge.getKonger("Frede")) print(konge);
}

private static void print(Konge konge)
{
 if (!konge.equals(Konge.NON))
  System.out.println(
   konge.getNavn() + ", " + konge.getFra() + " - " + konge.getTil());
}
```

There is not much to explain, but you should note that the type's methods are used as methods in other classes.

EXERCISE 5

Write a program that you can call *MonthProgram*. Add an enumeration named *Months* to the program when the type must represent the months as an enumeration, and when for each month must be attached an argument that shows the number of days in the month.

The type must beyond a *get* method for the number of days have a *static* method that returns the number of days in the year when the method has to be implemented with a loop that determines the number of days by iterating over the values of *Mounts*.

There must also be a *toString()*, that returns a string with the name of the month and the number of days in brackets.

Write a test method – the *main()* method – which just prints the number of days in the year.

There is a particular problem with initializing February correct. Here you should remember the class *GregorianCalendar*.

We have ambitions. Also for you.

SimCorp is a global leader in financial software. At SimCorp, you will be part of a large network of competent and skilled colleagues who all aspire to reach common goals with dedication and team spirit. We invest in our employees to ensure that you can meet your ambitions on a personal as well as on a professional level. SimCorp employs the best qualified people within economics, finance and IT, and the majority of our colleagues have a university or business degree within these fields.

Ambitious? Look for opportunities at www.simcorp.com/careers

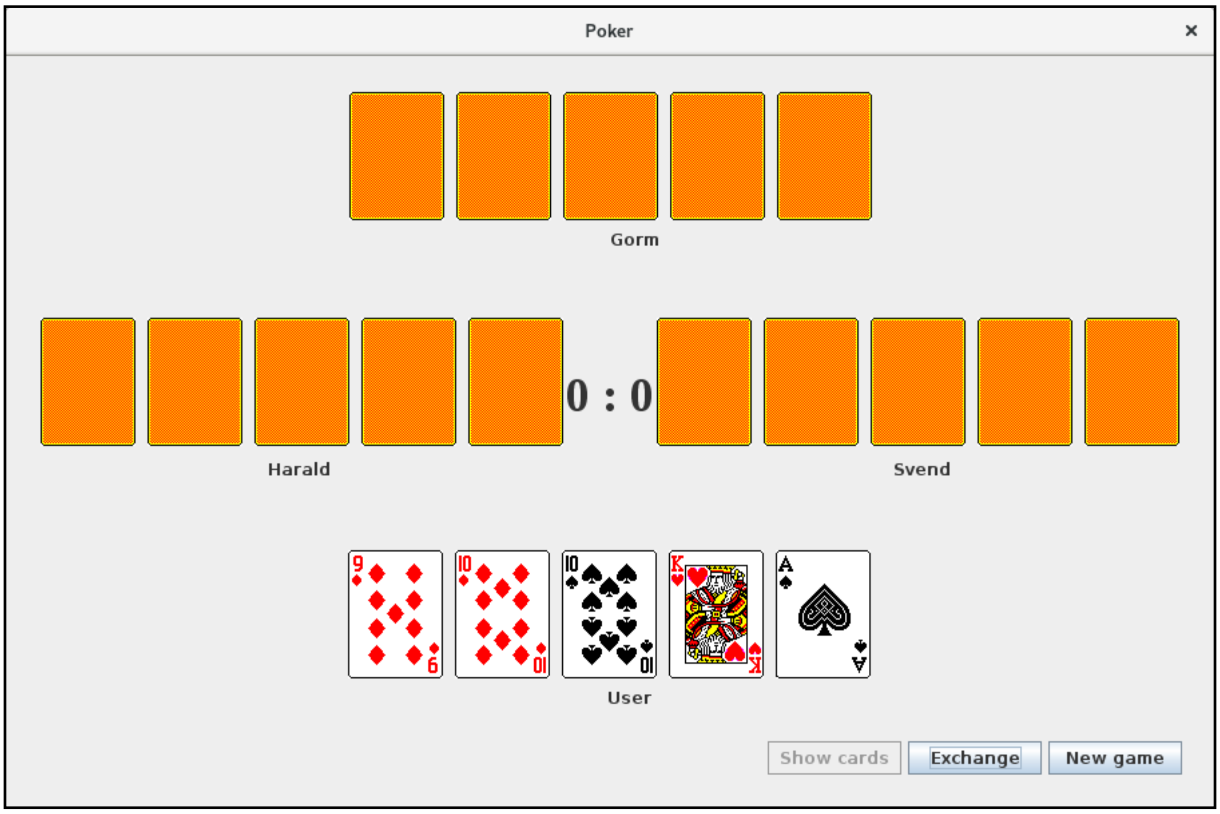
SimCorp

www.simcorp.com

PROBLEM 2

In this task, you must write a program that simulates that four players are playing poker. If you do not know the rules of poker, they are simple and are explained below. The purpose is to show examples of the use of enumerations, and it is recommended that you follow the procedure below. The program is intended as a vary simplified poker program.

The program must operate in the following manner. There are four players, one of which is the user, while the three others are virtual players. When starting a new game, each player is dealt 5 cards. The user can see his own cards but not the three others. An example of the program's window is:



After the players has there 5 cards, each player has the opportunity to exchange cards (possibly none or all cards), and once they have done it, you can view the cards and the program tells you who has won. You can then choose a new game, and in the middle the program shows, how many games you have played and how many the you have won.

The program's main shortcomings (restrictions) are as listed below:

1. There is nothing in deposits, and it makes it very uninteresting to play, as it is the whole idea of poker, and the ability to bluff falls thus also away.
2. The rules as the three virtual players use to exchange cards is very simple and static.
3. There are always four players, which should be more dynamic.
4. There is always played with 52 cards.

In addition, the user interface is simple. The task has these limitations to make the program simple. In fact, it is quite complex two solve the first two of the above constraints in a good way. I will later returns to the task and look at a new version of the program that do not to have the same restrictions, but in this place you has to focus on types and especially enumerations and write the program as described below.

In poker each player is dealt 5 cards, and the player that finally (after players have exchanged cards) has the highest value, has won the game. The values are with decreasing value downwards (where an ace is always the highest card):

- *straight flush*, where you have five subsequent cards of the same color, that is, for example eight, nine, ten, jack and queen in the same color
- four of a kind where you have four cards of the same value, that is, for example four kings
- full house, where you have a pair and three of a kind, for example two jacks and three fours
- *flush*, where you have five cards of the same color
- *straight*, where you have five subsequent cards, but not necessarily in the same color
- three of a kind where you have three cards with the same value, for example three sixes, but the last two cards are different and does not have the value six
- two pairs that is, for example two threes and two aces, while the last card is not three or an ace
- a pair where there are two cards with the same values such as two jacks, while the last three cards are not a jack, not three of a kind or contains a pair
- high card, where none of the above values occur

In the case that two players have the same value, the card with the highest value is determining, and are they the same (as an example two players with a pair), it is the value of the first card, where the two players' cards are different which determines the order.

Use the following procedure to write the program:

1)

Create a project that you can call *CardProgram*. Create a package *cardprogram.images* and copy the images of the cards to this package. The cards can be found in the file *cards.tar.gz*.

Add a class called *MainView* when it should be the program's main window and write so much code that the constructor opens a (so far empty) window. Creates an object of the class in *main()* and run the program, the result must be that the program opens a blank window.

2)

Add the following types to the project, representing respectively the color and the value of a playing card:

```
package cardprogram;

public enum Color { DIAMOND, HEART, SPADE, CLUBS }
```

What do you want to do?
No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!
VOLVO
AB Volvo (publ)
www.volvogroup.com
Volvo Trucks I Renault Trucks I Mack Trucks I Volvo Buses I Volvo Construction Equipment I Volvo Penta I Volvo Aero I Volvo IT
Volvo Financial Services I Volvo 3P I Volvo Powertrain I Volvo Parts I Volvo Technology I Volvo Logistics I Business Area Asia

```
package cardprogram;

public enum Value
{ TWWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
```

Note that the last type where the values are sorted, and an Ace has the highest value, what is the case in poker.

3)

Add a class representing a playing card, when the class should have two instance variables of the type *Color* and *Value* and when a playing cards must be *Comparable*:

```
package cardprogram;

/**
 * Represents a playing card by a color and a value.
 */
public class Card implements Comparable<Card>
{
 private Color color;
 private Value value;

 …

}
```

The class must override *equals()*, then two cards are equal if they have the same color and same value, and where two cards should be sorted only by their value.

Next, write a class *Cards* that represents a deck of 52 cards, and where the constructor creates the 52 *Card* objects:

```
package cardprogram;

import java.util.*;

public class Cards
{
 …

 public Cards() { }
 public boolean empty() { }

 public Kort deal() throws Exception { }

 public void shuffle() { }
}
```

The class is little more than an array of 52 cards. The class must have two important methods:

1. *shuffle()*, which shuffles the cards in random order.
2. *deal()*, which take the top card from the deck and is used to provide a player a card.

4)

Add a type that defines the size of a poker hand:

```
package cardprogram;
public enum Rank { HEIGHT_CARD, ONE_PIAR, TWO_PAIR, THREE_OF_A_KIND,
 STRAIGHT, FLUSH, FULL_HOUSE, FOUR_OF_A_KIND, STRAIGHTFLUSH }
```

where the order defines the size of a poker hand.

Add a class that defines a poker hand:

```
package cardprogram;

import java.util.*;

public class Hand implements Comparable<Hand>
{
 private Card[] arr = new Card[5]; // the cards
 private CardValues value; // represents the value of the hand

  /**
   * Creates a hand by taking 5 cards from the deck.
   * The 5 cards are arranged in increasing order of the cards value.
   * Then the value of the hand is determined and stored in the variable value.
   * @param cards The deck
   * @throws Exception If the deck does not contains 5 cards
   */
 public Hand(Cards cards) throws Exception {}
  public Card getCard(int n) {}

  /**
   * Exchange cards (choose new cards for some of the cards).
   * After the cards are exchanged the hand's value is determined again.
   * @param cards The deck
   * @param selected Array indicating which cards (places) to be exchanged
   * @return true, if the deck enough cards and all wanted cards are exchanged
   */
  public boolean exchange(Cards cards, boolean[] selected) {}

  /**
   * @return The rank of the hand
   */
  public Rank getRank() {}
  /**
   * Returns a string consisting of the hand's rank and cards
   * @return This hand represented as a string
   */
  public String toString() {}
```

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

```
 /**
  * If the current object and the parameter are of different rank the
  * comparison is trivial.
  * Otherwise, it is the values of the cards, that determines the result.
  * @param haand The hand that compares with
  * @return -1 (<), 1 (>) or 0 (==)
  */
 public int compareTo(Hand hand) {}

 // Inner class to the value of a hand.
 private class CardValues
 {
   // the cards that are needed to compare two hands in the case that they have
  // the same rank
   private Value[] value;
  private Rank rank; // hand rank

  public CardValues(Rank rank, Value … value)
  {
   this.value = value;
   this.rank = rank;
  }

  public Value getValue(int n)
  {
  return value[n];
   }

 public Rank getRank()
  {
   return rank;
  }
 }
}
```

The class has an inner class called *CardValues*, which represents the value of a hand. The class has two variables, which partly indicates the hand's rank, and the values that should be used to compare two hands. These values must be used to determine the hand's value in the case that two players has a hand of the same rank.

5)

Write a class Player, which represents a player with a name and a hand:

```
package cardprogram;

public class Player implements Comparable<Player>
{
 private String name;
 private Hand hand = null;

 public Player(String name) {}

 public String getName() {}
 /**
  * @return The player's rank
  * @throws Exception If the player has not yet been dealt cards
  */
 public Rank getRank() throws Exception {}

 public Hand getHand() throws Exception {}

 /**
  * The user want to exchange cards
  * @param cards Deck of cards
  * @param selected Indicates which cards are to be exchange
  */
 public void exhange(Cards cards, boolean[] selected) {}

 /**
  * A player other than the user want to exchange cards.
  * The method select which cards to exchange.
  * @param cards Deck of cards
  */
 public void exchange(Cards cards) {}

 /**
  * Giving the player a hand of 5 cards
  * @param cards The deck of cards
  * @throws Exception If the deck not has 5 cards
  */
 public void deal(Cards cards) throws Exception {}
 /**
  * Comparing the cards for two players in relation to the rules of poker.
  * The comparison ensure that players are arranged in descending order of value
  * of cards
  * @param player The player to compares with
  * @return 1 (hand > player.hand) -1 (hand < player.haand) otherwise 0
 */
 public int compareTo(Player player) {}

 // Two players are equal if they have the same name.
 public boolean equals(Object obj) {}
}
```

6)

Write a class *Poker* representing a number of players who are playing poker:

```
package cardprogram;

/**
 * Class which represents a number of players who are playing poker.
 */
```

This e-book
is made with
SetaPDF

SETASIGN

PDF components for **PHP** developers

www.setasign.com

```
public class Poker
{
 private Cards cards = new Cards();
 private Player[] players;

 /**
  * Creates a poker game with a deck of cards and a number of players.
  * @param players The players participating in the game.
  */
 public Poker(Player … players) {}

 /**
  * Starts a new game by shuffle the deck of cards and all players gets
  * there cards.
  * @throws Exception If there are not enough cards to all players.
  */
 public void deal() throws Exception {}

 /**
  * @return Number of players
  */
 public int size() {}

 /**
  * The player with index n.
  * @param n Index
  * @return The player with index n
  */
 public Player getPlayer(int n) {}

 /**
  * Exchange cards for the n-th player.
  * @param n Index for the player to exchange cards
  * @param selected Specifies which card (places) to be exhanged
  */
 public void exchange(int n, boolean[] selected) {}

 /**
  * Exchange cards for the n-th player.
  * @param n Index for the player to exchange cards
  */
 public void exchange(int n) {}
```

```
  /**
   * Determines which player has won and then the player who has the highest cards.
   * @return The player who has won
   */
 public Player won() {}
}
```

It's a simple class, and the most important method is the last, which returns the player who has won.

7)

Add the following type to keep track of who is who:

```
package cardprogram;
/**
 * Represents four players as their position at the table.
 */
public enum Orientation { NORTH, SOUTH, WEST, EAST }
```

Add a controller class that must keep track of how many games are played and how many the user has won, and treat events from the three buttons in the user interface:

1. exchange cards
2. find the winner
3. new game

With this class in place all that remains is to write the code to the application's *MainView*.

6 EXCEPTION HANDLING

In the preceding examples, both in this and the previous books I have used exception handling. In principle, it is quite simple, but still are a few more things you should know about.

In practice it is often that a method of one reason or another can not perform the desired operation, for example because the parameters have illegal values, or otherwise an error occurs when the method is performed. When the method acknowledges the error, it may of course cancel the action, but it must also make the calling code aware that there is an error. This can be achieved by let the method raises an exception, or it can be done with a return value. The last was formerly a widely used method for handling these kinds of errors, and the way has both its applications and limitations, so I will start there. I will use the project *Students* from the previous book, where I will introduce some changes in order to better error handling.

"I studied English for 16 years but... ...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

I have extended the class *Student* with a method that deletes a course, and the method is defined in the interface *IStudent* as follows:

```
/**
 * Deletes a course which a student is enrolled or have completed.
 * @param course The course to be deleted
 * @return true, if the course was deleted and false otherwise
 */
public boolean remove(ICourse course);
```

A course can obviously only be dropped if the student have the course in question. The method can fail, and if so, the calling code has to be notified. It could be solved by changing the method so it raises an exception, but here the method instead returns a value (*true* or *false*) indicating whether the method is performed correctly or not. The method must therefore have a return value, which tells the users how it was accomplished. In this case, it is simple to implement the method as the class *ArrayList* work the same way:

```
public boolean remove(ICourse course)
{
 return courses.remove(course);
}
```

A prerequisite for the method works is, moreover, that the class *Course* overload *equals()* with value semantics, and this is not the case and you have to implements *equals()* in the class *Course*:

```
public boolean equals(Object obj)
{
 if (obj == null) return false;
 if (getClass() == obj.getClass())
  return getId().toUpperCase().equals(((Course)obj).getId().toUpperCase());
 return false;
}
```

The use of the return values as an indication of whether a method was properly performed is an excellent solution and has the advantage that it is not necessary to encapsulate the called code in a try/catch. However, it also has its limitations or drawbacks, if the method returns a value. Where appropriate, the return value may be incorrectly interpreted as a result of a return value rather than an error code. Another problem is that the user can simply ignore the return value, and do not test it.

The class *Course* has a method that returns the score that a student has achieved in the subject:

```
public int getScore() throws Exception
{
 if (score == Integer.MIN_VALUE)
  throw new Exception("The student has not completed the course");
 return score;
}
```

If the students do not yet have a score the method raises an exception. That a student has not received a score is known by the variable of *score* that has the value *Integer.MIN_VALUE* (which is -2147483648), and it is unlikely to be a legal score. The method could, therefore, simpler be written as follows:

```
public int getScore()
{
 return score;
}
```

where the value -2147483648 then has to be interpreted as an error code, and it would mean that the method could be used without having to be placed in a try/catch. On the other hand, you would as a mistake could be using the error code as a score, with the result that a calculation could be wrong. Another consequence would be that the user in most cases has to test the value and examine whether it is an error code or a legal score. In this case, I prefer to keep the method as it is, and let it raise an exception if there has not yet been assigned a score.

If you have a method that returns an object as such the following method from the class *Student*

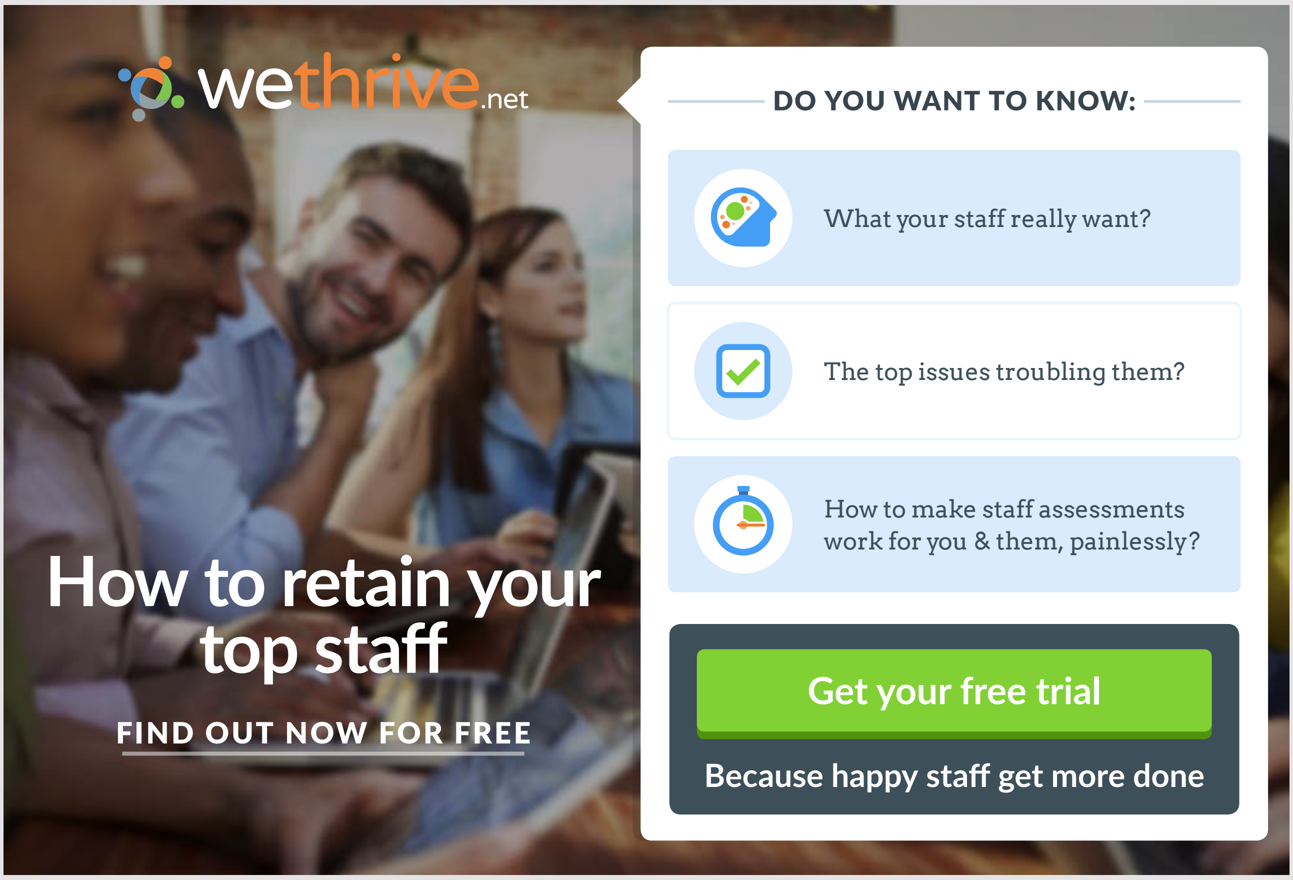
```
public ICourse getCourses(String id) throws Exception
{
 for (ICourse c : courses) if (c.getId().equals(id)) return c;
 throw new Exception("Course not found");
}
```

it can raise an exception, because the students do not necessarily have a course with the current id. Sometimes, and perhaps even quite often you see this problem solved by the method in the case that there is no course returns *null*:

```
public ICourse getCourses(String id)
{
 for (ICourse c : courses) if (c.getId().equals(id)) return c;
 return null;
}
```

This solution is used very often, when it is clear that *null* means an error or indicate that something was not found, but here too there is a risk that the return value could be used as an object when it was *null*, and the result would be a *NullPointerException*.

The conclusion is that the use of return values as error messages has its uses, but in most cases it is better to let the method raise an exception.


wethrive.net
DO YOU WANT TO KNOW:
What your staff really want?
The top issues troubling them?
How to make staff assessments work for you & them, painlessly?
Get your free trial
Because happy staff get more done
How to retain your top staff
FIND OUT NOW FOR FREE

6.1 CHECKED EXCEPTIONS

The principle in exception handling is that a method can fail, and if so, it raises an exception, which means that the method immediately is interrupted with a message to the method that called it. Called a method that can raise an exception, the calling code must handle this exception, which typically consists of encapsulating the method call in a try/catch. Otherwise, the calling code is sending an exception further up in the hierarchy of calling methods. The idea is that the method that can raise an exception can see that something is wrong, but the method can not know how the error should be handled. It can, however, send an exception to the calling code in the hope that it knows how the error should be handled, and does it not that, it can forward the message up in method hierarchy.

Above, I have everywhere let methods raise an exception of the type *Exception*, and although it works, it's not the idea. Javas own classes raises many exceptions and they have types that tell us something about what is the reason for the exception. These exception types are all directly or indirectly derived from the class *Exception*, and although one probably can use some of these types of exceptions, it is typical that you in the context of an application defines its own exception types. The reason is partly that in this way you can classify the various exceptions and treat them differently but also that you get the chance to send values to the calling code.

In the project concerning students I have added the following class:

```
package students;

public class StudentsException extends Exception
{
 public StudentsException(String message)
 {
  super(message);
 }
}
```

It is an exception type for exceptions for this program. You should note that the class does not add anything new, and its sole purpose is to characterize the exceptions raised by methods of the program's classes. In this way, these exceptions are distinguished from exceptions raised by Java's classes. I also added the following exceptiontype which defines the exceptions to the class *Subject*:

```
package students;

class SubjectException extends StudentsException
{
 private String field;

 public SubjectException(String message, String field)
 {
  super(message);
  this.field = field;
 }

 public String getField()
 {
  return field;
 }
}
```

The class *Subject* raises an exception if one of its fields are assigned an illegal value. Therefore extends this class *StudentsException* with a variable that indicates which field that is illegal. Also note that the class is written in the same file as *StudentsException* and thus has package visibility. It was chosen because it is a very simple class, and the type should not be known outside of the program's package. With these types the class *Subject* must be updated as follows, where I have only shown the methods that raises an exception:

```
public class Subject implements ISubject, IPoint, Cloneable
{
 private String id; // the subject id
 private String name; // the subject's name
 private int ects = 0; // the subtect's ECTS

 public Subject(String id, String name) throws SubjectException
 {
  this(id, name, 0);
 }

 public Subject(String id, String name, int ects) throws SubjectException
 {
  if (!ISubject.subjectOk(id, name)) throw new
   SubjectException("The subject must have both an ID and a name", "name");
  if (ects < 0) throw new SubjectException("ECTS must be non-negative", "ects");
  this.id = id;
  this.name = name;
  this.ects = ects;
 }
 public void setName(String name) throws SubjectException
 {
  if (!ISubject.subjectOk(id, name)) throw new
   SubjectException("The subject must have a name", "name");
  this.name = name;
 }

 public void setECTS(int ects) throws SubjectException
 {
  if (ects < 0) throw new SubjectException("ECTS must be non-negative", "ects");
  this.ects = ects;
 }
}
```

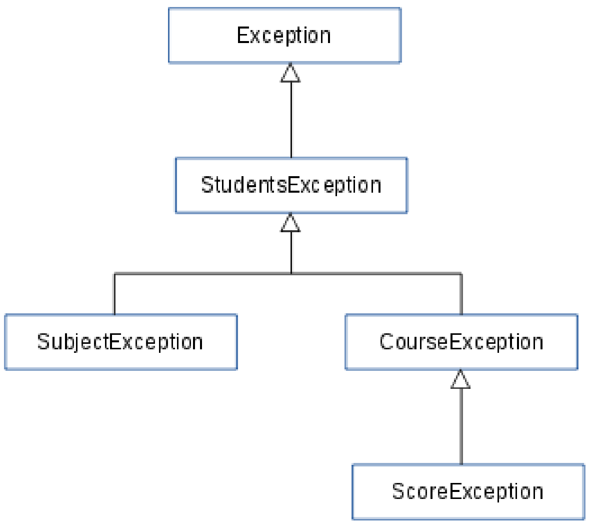
Note that also the interface *ISubject* and *IPoint* must be updated such that the methods throws the correct exception types. Also note that the program can compile and run, although the test methods referes to *Exception* instead of *SubjectException*, for a *SubjectException* is especially an *Exception*.

For the sake of courses I have defined the following exception types:

```
class CourseException extends StudentsException
{
 public CourseException(String message)
 {
  super(message);
 }
}

class ScoreException extends CourseException
{
 private int score;

 public ScoreException(String message, int score)
 {
  super(message);
  this.score = score;
 }

 public int getScore()
 {
  return score;
 }
}
```

Next, the class *Course* should be updated with these types, and the same applies to the interface *ICourse.*

On this place I have defined four exception classes which form a hierarchy:



It is very common that exception types for a program in this way consists of a hierarchy of classes with *Exception* as a common base class. On the other hand, it depends on the job, programmer, etc., how far you go and how many types you define, and above I have probably gone to far which the benefits of the many types is modest.

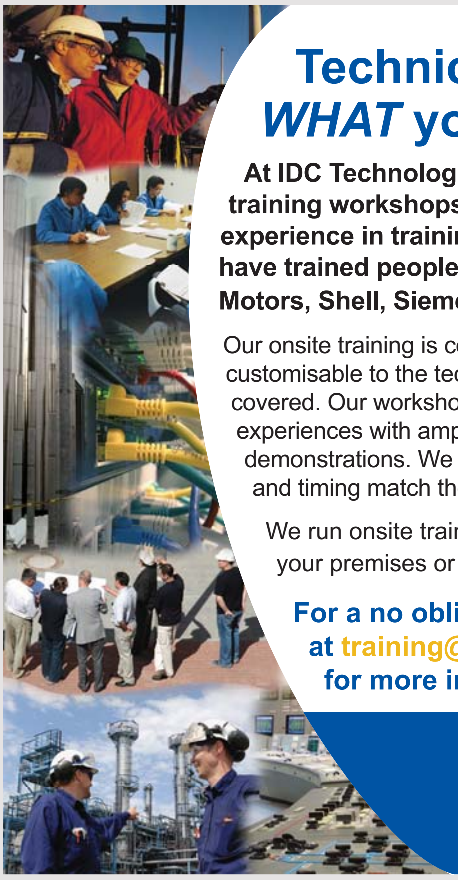
I've also updated four of the methods in the class *Factory*:

```
public abstract class Factory
{
 public static ISubject createSubject(String id, String name)
  throws SubjectException
 {
  return new Subject(id, name);
 }

 public static ISubject createSubject(String id, String name, int ects)
  throws SubjectException
 {
  return new Subject(id, name);
 }

 public static ICourse createCourse(int year, ISubject subject)
  throws CourseException
 {
  return new Course(year, subject);
 }

 public static ICourse createCourse(int year, String id, String name)
  throws CourseException, SubjectException
 {
  return new Course(year, createSubject(id, name));
 }
```

The four methods return new exceptions similar to those exceptions the concrete constructors can raise. You should especially notice the last. When *createSubject()* can raise an exception of the type *SubjectException*, the method can raise two kinds of exceptions and you need to note how you with *throws* has to indicate that a method can raise several kinds of exceptions.

When a method with *throws* indicates that it may raise an exception, the compiler demands that the calling code must be placed in a try/catch. Thereby forcing the programmer to deal with the exceptions that a method can raise. It may seem cumbersome, but it helps to provide a robust code. Is there anything that can result in an error, then the programmer must necessarily consider what to do if the error occurs. It's the whole fundamental principle of exception handling. Below is a test method:

```
private static void test()
{
 try
  {
   ICourse course1 = Factory.createCourse(2015, "MAT7", "Mathematics");
   course1.setScore(7);
   System.out.println(course1.getId());
   System.out.println(course1 + " " + scourse1.getScore());
   ICourse course2 = Factory.createCourse(2015, "MAT6", "");
   System.out.println(course2.getSubject());
  }
  catch (SubjectException ex)
  {
   System.out.println(ex.getField() + ": " + ex.getMessage());
  }
```



Technical training on
***WHAT* you need, *WHEN* you need it**

At IDC Technologies we can tailor our technical and engineering training workshops to suit your needs. We have extensive experience in training technical and engineering staff and have trained people in organisations such as General Motors, Shell, Siemens, BHP and Honeywell to name a few.

Our onsite training is cost effective, convenient and completely customisable to the technical and engineering areas you want covered. Our workshops are all comprehensive hands-on learning experiences with ample time given to practical sessions and demonstrations. We communicate well to ensure that workshop content and timing match the knowledge, skills, and abilities of the participants.

We run onsite training all year round and hold the workshops on your premises or a venue of your choice for your convenience.

For a no obligation proposal, contact us today at training@idc-online.com or visit our website for more information: www.idc-online.com/onsite/

OIL & GAS ENGINEERING

ELECTRONICS

AUTOMATION & PROCESS CONTROL

MECHANICAL ENGINEERING

INDUSTRIAL DATA COMMS

ELECTRICAL POWER

Phone: **+61 8 9321 1702**
Email: **training@idc-online.com**
Website: **www.idc-online.com**

IDC TECHNOLOGIES

```
   catch (ScoreException ex)
  {
   System.out.println(ex.getMessage());
  }
   catch (CourseException ex)
  {
   System.out.println(ex.getMessage());
  }
   finally
  {
   System.out.println("End!!");
  }
}
```

The code in the try block can raise three kinds of exceptions, and there is, therefore, three catch blocks. It is the type of a possible exception, that determines which catch is performed and it is one of the main reasons to have more exception types because in this way you can control the action that should happen. In this case there is no particular reason for it, since the error handler each time consists of printing an error message. Note, however, the first catch, where I use, that the exception has a method *getField()*. You should also notice that the handler for *ScoreException* must come before the handler for *CourseException* because a *ScoreException* also is a *CourseException* and the runtime system looks for a match from start to bottom.

Finally, there is a *finally* block. It does not have to be there, and in this case, it has no sense, but a finally block is always performed regardless of whether there is an exception or not. It may, for example be used to close files, terminate connections to databases etc. If the method is executed the result is:

```
2015-MAT7
Mathematics 7
name: A subject must have a name
End!!
```

Here you should note that the last statement in the try block is not executed because it raises an exception and thus interrupt the try block. Note also that the *finally* block is performed.

As mentioned, there in this example is no particular need to distinguish between the three types of exceptions, and the code could instead be written as follows:

```
private static void test()
{
 try
 {
   ICourse course1 = Factory.createCourse(2015, "MAT7", "Mathematics");
   course1.setScore(7);
   System.out.println(course1.getId());
   System.out.println(course1 + " " + scourse1.getScore());
   ICourse course2 = Factory.createCourse(2015, "MAT6", "");
   System.out.println(course2.getSubject());
 }
 catch (StudentsException ex)
 {
   System.out.println(ex.getMessage());
 }
 finally
 {
   System.out.println("End!!");
 }
}
```

for all three kinds of exceptions's are also a *StudentsException*.

As discussed above a user-defined exception directly or indirectly inherit *Exception*, and if a method that can raise such an exception is performed, you must with *throws* specify which exception the method can raise. Such exceptions are said to be *checked*.

6.2 UNCHECKED EXCEPTIONS

There is another possibility. I have extended the class *Team* with a method that returns the n-th student:

```
public IStudent getStudentAt(int n) throws Exception
{
  if (n < 0 || n >= list.size()) throw new Exception("Illegal index for Student");
  return list.get(n);
}
```

It raises an exception if *n* is an illegal index. In a way, it is not interesting, because if *n* is illegal, the method will automatically raise an exception, which is a *RuntimeException*, and it is an exception, which need not necessarily be handled. If you still even need to capture the index error (possibly because of the error message) you could write the method as follows:

```
public IStudent getStudentAt(int n)
{
  if (n < 0 || n >= list.size())
    throw new RuntimeException("Illegal index for Student");
  return list.get(n);
}
```

so that it now raises a *unchecked* exception. Note that this means that the method no longer has a *throws*, and that in turn means that the method can be used outside of a try/catch.

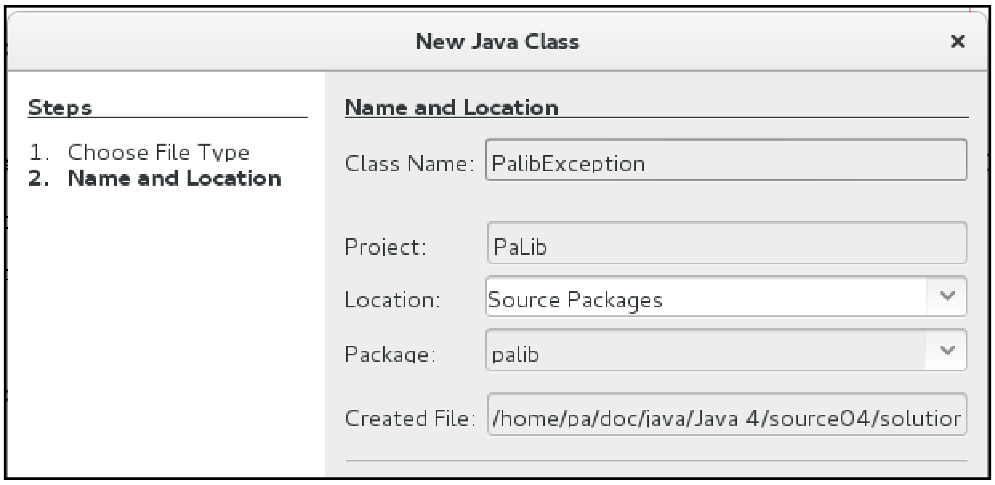
If you in your own programs uses unchecked exceptions, you should in the same as with checked exceptions define your own exception types that are classes derived directly or indirectly from *RuntimeException*. You must remember, however, that there are rarely good reasons to use unchecked exceptions. They are intended for exceptions regarding serious errors where it no longer make sense to continue the program's execution. Applied unchecked exceptions in your own programs, you override just the whole idea of exception handling where you force the calling code to treat any exceptions, but there are situations, and I will later show examples when I look on abstract data types.

EXERCISE 6

Open the project *PaLib* again. The project consists for now of two packages

1. *palib.util*
2. *palib.gui*

but there is also a third package, called *palib* – although NetBeans does not show it. You must add an exception type called *PalibException* when it should be in the package *palib*. Note that you must enter the name of the package:


New Java Class

Steps
1. Choose File Type
2. Name and Location

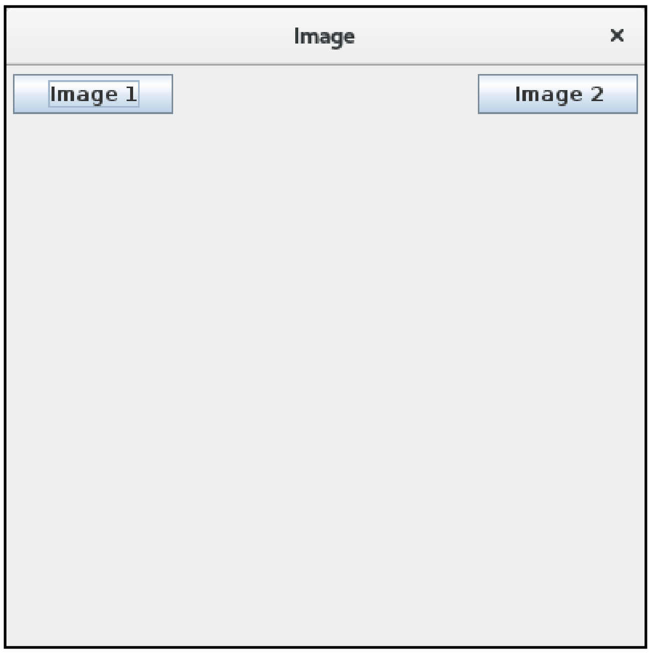
Name and Location
Class Name: PalibException
Project: PaLib
Location: Source Packages
Package: palib
Created File: /home/pa/doc/java/Java 4/source04/solutior

You must then add an exception class in the package *palib.gui*, when the class is called *GuiException* and when it must be derived from *PalibException*.

The class *Tools* has a method *createImageIcon()* which loads an image from the application's jar file. If the image can not be loaded (may be because the name is wrong) the method returns *null*, which is perhaps an unfortunate solution. You must change the method so that it instead raises a *GuiException* if an error occurs while loading the image. Remember that you must also update the comment. Build the project so it is ready for use.

Create a new program, that you can call *ImageProgram*. Add a package to the project with the name *imageprogram.images*. Copy the image *stone.jpg* (from this books directory) to the new package. Add a reference to the class library *PaLib* to the project.

The program must open the following window, where the center contains a label:

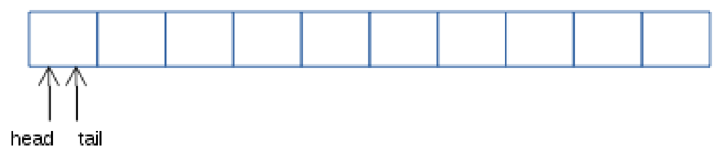


When the user click on one button, the program should show the image *stone.jpg*, an when the user click on the other button, the result should be a *GuiException* (for example, because the name is misspelled).

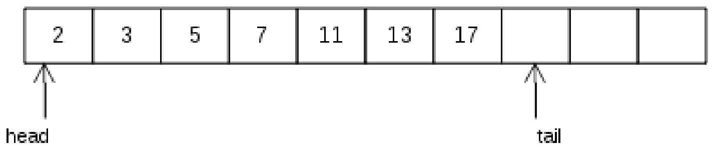
7 GENERIC TYPES

In this chapter I will look at generic types and generic methods. In fact, I already used generic types many times as such an *ArrayList<T>*, which indicate the type of the kind of objects that the list should contain. Similarly, I have used several generic interfaces such as *Comparable<T>*. In this section I will show how also user-defined types can be defined generic.

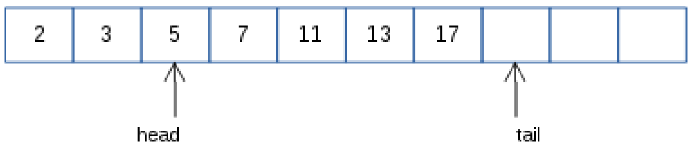
I would start with a class representing a circular buffer. A buffer is a container (with Java terms a collection), which may contain objects, and to the buffer is associated methods that manipulate the content. A buffer can be implemented in several ways, but here I will look at a so-called circular buffer, a buffer with room for a certain number of objects. Such a buffer can be easily implemented by means of an array:



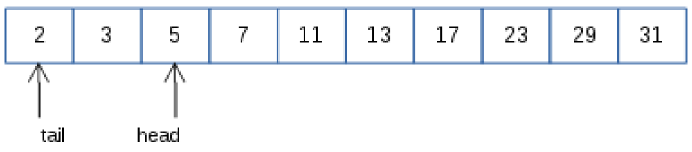
Basically, there are to a buffer associated two operations called *insert()* and *remove()*, where the first inserts an element into the buffer, while the other removes the oldest element – the element that has been longest in the buffer. The figure above illustrates an empty buffer, and the arrow *tail* is the index where the next element should be inserted. Similarly, the *head* is the index of the next element to be removed. Initially, the buffer is empty and the arrows points to the same place. Below is how the buffer looks after inserted 7 elements – the method *insert()* is executed 7 times:



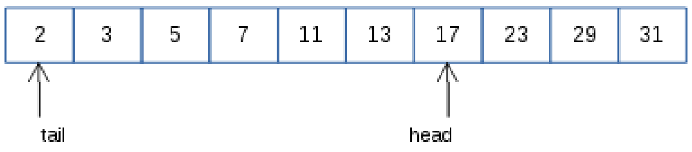
If you then remove two elements – the method *remove()* is executed 2 times – the picture is:



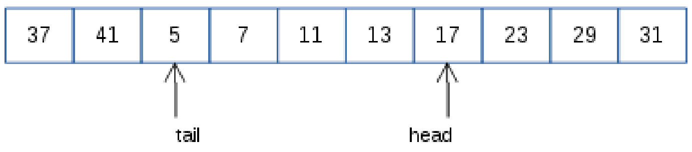
The elements are in principle remained in the data structure, but there is no access to them. If you then add 3 elements, you get the result:



Here you should especially note that the *tail* index wraps around so that the next element to be inserted is in position 0. The next figure shows the buffer, after 4 additional elements are removed:



and the latter figure shows the result after insertion of two additional elements:



The name *circular buffer* is derived from the indices that wraps around when they reach the end of the array.

It is simple to implement such a data structure, and the first and foremost challenge is to keep track of when the buffer is full and empty. Of course one can not add elements to a full buffer, just as you can not remove elements from an empty buffer. The problem can be solved in several ways, but I will use a simple counter that counts the number of elements. This requires an additional variable, but it is, in turn, easy and efficient to implement each method, as it is easy to ask the buffer about the number of elements it contains. Following these considerations, you can implement a buffer to integers as follows:

```
package generic;

public class IntBuffer
{
 private int[] buff; // array to the buffer
 private int head = 0; // index to the place in front of the first element
 private int tail = 0; // index of the last element of the buffer
 private int count = 0; // number of elements in the buffer

 public IntBuffer(int n)
 {
  buff = new int[n];
 }

 public int getCount()
 {
  return count;
 }

 public boolean empty()
 {
  return count == 0;
 }

 public boolean full()
 {
  return count == buff.length;
 }

 public int peek() throws Exception
 {
  if (empty()) throw new Exception("The buffer is empty");
  return buff[head];
 }
 public int remove() throws Exception
 {
  if (empty()) throw new Exception("The buffer is empty");
  int elem = buff[head];
  head = next(head);
  --count;
  return elem;
 }

 public void insert(int elem) throws Exception
 {
  if (full()) throw new Exception("The buffer is full");
  buff[tail] = elem;
  tail = next(tail);
  ++count;
 }

 private int next(int n)
 {
  return (n + 1) % buff.length;
 }
}
```

There is not much to explain, and the implementation is quite effective. The two methods *empty()* and *full()* is trivial and the fundamental methods *insert()* and *remove()* are also effective, since in addition to test for an exception they solely do some simple operations. You should also note the method *peek()*, which returns the oldest element in the buffer, but without removing it.

As an example the method *test01()* creates an *IntBuffer* with room for five numbers. The program will iterate over a loop where it randomly either insert a number or remove a number. Some operations will fail because the buffer is either full or empty.

```
private static void test01()
{
  IntBuffer buffer = new IntBuffer(5);
  for (int i = 0; i < 100; ++i)
   if (rand.nextBoolean())
    try
    {
     System.out.println("<< " + buffer.remove());
    }
    catch (Exception ex)
    {
     System.out.println(ex.getMessage());
    }
   else
    try
    {
     int t = rand.nextInt(90) + 10;
     buffer.insert(t);
     System.out.println(">> " + t);
    }
    catch (Exception ex)
    {
     System.out.println(ex.getMessage());
    }
}
```

The class *IntBuffer*, however, has one disadvantage, as it only can be used for elements of type *int*. To use a buffer to other types of elements, it is necessary to add a new class. One solution is to write a buffer where the element type is *Object*:

```
package generic;

public class ObjBuffer
{
  private Object[] buff; // array to the buffer
  private int head = 0; // index to the place in front of the first element
  private int tail = 0; // index of the last element of the buffer
  private int count = 0; // number of elements in the buffer

 public ObjBuffer(int n)
 {
  buff = new Object[n];
 }

 public int getCount()
 {
  return count;
 }

 public boolean empty()
 {
  return count == 0;
 }

 public boolean full()
 {
  return count == buff.length;
 }

 public Object peek() throws Exception
 {
  if (empty()) throw new Exception("The buffer is empty");
  return buff[head];
 }

 public Object remove() throws Exception
 {
  if (empty()) throw new Exception("The buffer is empty");
  Object elem = buff[head];
  head = next(head);
  --count;
  return elem;
 }
 public void insert(Object elem) throws Exception
 {
  if (full()) throw new Exception("The buffer is full");
  buff[tail] = elem;
  tail = next(tail);
  ++count;
  }

 private int next(int n)
 {
  return (n + 1) % buff.length;
 }
}
```

You should note that it is almost the same class, and it is only a question of the type of the elements everywhere is changed from *int* to *Object*. All the algorithms are exactly the same and have the same effectiveness as in the class *IntBuffer*. With this buffer you can write something like the following:

```
private static void test03()
{
 ObjBuffer buffer = new ObjBuffer(10);
 try
 {
  buffer.insert("Svend");
  buffer.insert(23);
  buffer.insert("Knud");
  buffer.insert(3.14);
  buffer.insert("Valdemar");
  buffer.insert(new IntBuffer(5));
  while (!buffer.empty()) System.out.println(buffer.remove());
 }
 catch (Exception ex)
 {
  System.out.println(ex.getMessage());
 }
}
```

and if the method is executed the result is

```
Svend
23
Knud
3.14
Valdemar
generisk.IntBuffer@6d06d69c
```

The buffer can contain everything that is an *Object*, and in this case it is a *String*, an *Integer*, a *String*, a *Double*, a *String* and an *IntBuffer*. Note that the compiler performs auto boxing of 23 and 14.3. It immediately seems smart, and in this case it is also fine as the only thing to do with the items when removed from the buffer is that they are printed – and any *Object* has a *toString()*. In other cases, however, it will be necessary to test the type of the elements when they are removed from the buffer and make a proper typecast. The biggest problem, however, is that the class *ObjBuffer* is not type strong, as you can fill anything into it, and it increases the risk of errors. This can be solved by defining a generic type.

Looking at the test method *test03()*, it is in fact that rarely that you needs a collection, which can contain anything. It is far more common that you need a collection of integers, a collection to strings or to objects of one type or another. To solve this problem, it is possible to parameterize a class where a parameter specifying the type of the objects that the class has to work with. Such a parameterized class is called a generic class, and the class buffer may be written as follows:

```
package generic;

import java.util.*;

public class Buffer<T> implements IBuffer<T>
{
 private T[] buff; // array to the buffer
 private int head = 0; // index to the place in front of the first element
 private int tail = 0; // index of the last element of the buffer
 private int count = 0; // number of elements in the buffer
 public Buffer(int n)
 {
  buff = Utils.createArray(n);
 }

 public int getCount()
 {
  return count;
 }

 public boolean empty()
 {
  return count == 0;
 }

 public boolean full()
 {
  return count == buff.length;
 }

 public T peek() throws Exception
 {
  if (empty()) throw new Exception("The buffer is empty");
  return buff[head];
 }

 public T remove() throws Exception
 {
  if (empty()) throw new Exception("The buffer is empty");
  T elem = buff[head];
  head = next(head);
  --count;
  return elem;
 }
 public void insert(T elem) throws Exception
 {
  if (full()) throw new Exception("The buffer is full");
  buff[tail] = elem;
  tail = next(tail);
  ++count;
 }

 private int next(int n)
 {
  return (n + 1) % buff.length;
 }
}
```

First, observe that this version of the buffer is nearly identical to the previous two, and that it is primarily a question of the element type anywhere is replaced by a parameter *T*. When the class is declared, you specify that it is generic:

```
public class Buffer<T>
```

and depends on a single parameterized type *T*. Throughout the class *T* is used as it was a concrete type. The compiler of course do not know what T is, and the only thing you can do with a *T* object is also what you can do with an *Object*, but for collection classes such as *Buffer* is also the only thing that is needed. The only problem is the creation of the array, since you can only create an array whose type is *Object* – you can not write for example *new T[99]*, since the compiler does not know what *T* is for a type. As I often will need to create a generic array, I have moved this operation to a method in a class *Utils* to be able to use it in other contexts:

```
class Utils
{
 public static <T> T[] createArray(int length, T… arr)
  {
   return Arrays.copyOf(arr, length);
  }
}
```

Preliminary simply accept the syntax, but *copyOf()* is a static method in the class *Arrays*, which creates an array of a given length, and copy another array. Note that the class *Utils* is defined in the same file as the class *Buffer*, but it should be placed somewhere else, what is the topic for the next exercise.

The class *Buffer<T>* implements an inteface:

```
public class Buffer<T> implements IBuffer<T>
```

and the interface does nothing more than define the class's methods. The aim is alone to show that also an interface may be generic:

```
package generic;

/**
  * Defines a generic buffer for objects of a certain type.
  */
public interface IBuffer<T>
{
 /**
  * @return Number of elements in the buffer
  */
 public int getCount();
```

```
  /**
   * @return true, if the buffer is empty
   */
  public boolean empty();

  /**
   * @return true, if the buffer is full
   */
  public boolean full();

  /**
   * Returns the oldest (first) element in the buffer without removing it,
   * @return The oldest (first) element of the buffer
   * @throws Exception If the buffer is empty
   */
  public T peek() throws Exception;

  /**
   * Returns the oldest (first) element in the buffer and remove the element from
   * the buffer.
   * @return The oldest (first) element of the buffer
   * @throws Exception Iff the buffer is empty
   */
  public T remove() throws Exception;

  /**
   * Adds an element to the buffer.
   * @param elem The element to be added
   * @throws Exception If the buffer is full
   */
  public void insert(T elem) throws Exception;
}
```

The advantage of generic types as *Buffer<T>* is that the compiler can test whether the type is used correctly, and any errors are localized, before a program is used. Therefore, it is not advisable to use types the likes *ObjBuffer* as they increase the risk that the generated code containing errors.

However, it should be noted that the type parameter to a generic class must be an *Object* and can not be a simple type. As an example you can not write:

```
Buffer<int> buffer = new Buffer(10);
```

and if you needs a buffer to the elements of a simple type, it is necessary to use wrapper classes:

```
Buffer<Integer> buffer = new Buffer(10);
```

However, it is not as big a problem as the compiler in many contexts perform the required type conversions by auto boxing and auto unboxing.

EXERCISE 7

In this exercise you shoul as in exercise 6 work on PaLib.

You must add a new exception class to *palib.util* named *UtilException*.

You must then copy the two types *IBuffer<T>* and *Buffer<T>* to *palib.util* and you must modify the types such that they everywhere (there are three methods) raises a *UtilException* instead of an *Exception*. Also remember to update the *IBuffer<T>* and remember to update the comments.

Add a class *Utils* (still in *palib.util*). The class shall initially only have the method *createArray()* from the corresponding class in the file with *Buffer<T>*, and then the class should be deleted in this file.

Build the class library.

Finally, write a test program called *TestBuffer* when this program must use your class library and perform the same as the test method *test01()* in the project *Generic* (see above).

7.1 MORE ON PARAMETERS

The name of the parameter type plays no role and the type can be anything and especially also another generic type. As an example the following method creates an *ArrayList* to elements of the type *ArrayList<Integer>*:

```
private static void test04()
{
 ArrayList<ArrayList<Integer>> lists = new ArrayList();
 lists.add(new ArrayList());
 lists.add(new ArrayList());
 lists.get(0).add(2);
 lists.get(0).add(3);
 lists.get(0).add(5);
 lists.get(0).add(6);
 lists.get(1).add(23);
 lists.get(1).add(29);
 for (ArrayList<Integer> list : lists)
 {
  for (Integer t : list) System.out.print(t + " ");
  System.out.println();
 }
}
```

and if you execute the method the result is:

```
2 3 5 6
23 29
```

As mentioned, you can specify multiple parameter types, and the class below shows a parameterized class *Pair* that is parameterized by two parameters:

```
package generisk;

public class Pair<K, V>
{
 private K key;
 private V value;

 public Pair(K key, V value)
 {
  this.key = key;
  this.value = value;
 }

 public K getKey()
 {
  return key;
 }

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

EXERCISE 8

Add the class *Pair<K, V>* to your class library when it must be added the package *palib.util*. Create a test program that defines an *ArrayList* with elements of type *Pair*, where the first parameter is a *String* (the name of a king), while the second parameter, is a *Pair<Integer, Integer>* and indicates the king's reign. Place the following kings in the data structure:

- Gorm den Gamle, 936, 958
- Harald Blåtand, 958, 987
- Svend Tveskæg, 987, 1014

and print the content of the data structure on the screen when the result must be printed in a method *print()*. The result should be

```
Gorm den Gamle, 936 - 958
Harald Blåtand, 958 - 987
Svend Tveskæg, 987 - 1014
```

7.2 RAW CLASSES

Java has a so-called collection API discussed later in this book, which contains generic classes. An *ArrayList* is an example, but a generic class may also be used as a so-called raw class in which you do not indicate a parameter type. Consider the following method:

```
private static void test05()
{
 ArrayList list = new ArrayList();
 list.add("Knud");
 list.add(3.14);
 list.add(new Pair<Integer, String>(1, "Margrete"));
 for (Object obj: list) print(obj);
 Buffer buffer = new Buffer(10);
 try
 {
  buffer.insert("Knud");
  buffer.insert(3.14);
  buffer.insert(new Pair<Integer, String>(2, "Margrethe"));
  while (!buffer.empty()) print(buffer.remove());
 }
 catch (Exception ex)
 {
   System.out.println(ex.getMessage());
  }
}

private static void print(Object obj)
{
  System.out.println(obj + ": " + obj.getClass());
}
```

This method creates an *ArrayList*, but without specifying a parameter. The compiler can not check the type of the objects added to the list, and it processes the list, as was the parameter of the type *Object*. This means that you can add anything to the list. The same applies to custom generic types, and you can create a *Buffer* without specifying a parameter type. If the above method is performed, you get the result:

```
Knud: class java.lang.String
3.14: class java.lang.Double
generisk.Pair@6d06d69c: class generisk.Pair
Knud: class java.lang.String
3.14: class java.lang.Double
generisk.Pair@7852e922: class generisk.Pair
```

Generally it is not advisable to use the raw versions of the generic classes since the compiler can not type check and leads to code that easily can contain errors.

7.3 GENERIC METHODS

Also, methods can be defined generic. Below is a method that seeks an element in an array:

```
public static <T> int linSearch(T[] arr, T elem)
{
  for (int i = 0; i < arr.length; ++i) if (arr[i].equals(elem)) return i;
  return -1;
}
```

The method is defined generic, as it acts on an arbitrary array. The algorithm is simple and consists only in a run through the array from start to finish. If the element exists, the method returns the element's index. If it is not found, the method returns -1, when it can not be a legally index. You should note that the method returns the index of the first element found. You should also note that the method implicitly assumes that the parameter type implements *equals()* with value semantic. An example of an application of the method could be:

```
private static void test06()
{
  Integer[] arr = { 2, 3, 5, 7, 11, 13, 17, 19 };
  System.out.println(Generic.<Integer>linSearch(arr, 11));
  System.out.println(linSearch(arr, 12));
}
```

Note that in the first call of the method I indicates the parameter type, which in principle is the correct syntax, but it is not necessary because the compiler from the type of the array *arr* can see that the parameter type is *Integer*.

I have previously shown the method *createArray()* and it is another example of a generic method.

EXERCISE 9

Add the method *linSearch()* to the class *Utils* in your class library.

Then you must write a program, that you can call *SearchProgram1*. The program must have a method

```
private static Integer[] createArray(int n)
{
}
```

that creates and returns an array with n elements, that must be the even numbers

2, 4, 6, 8, 10, …

and after the array is created and initialized the method should shuffle the elements, such they occur in random order.

There should also be a method

```
private static void search(Integer[] arr, int elem)
{
}
```

that search an element in the array with the library method *linSearch()*. The method should print where the element is found or not and how many nano seconds the search has taken.

In the *main()* method you should create an array with 10000000 elements (og the type *Integer*). Then you should run a loop, where you must enter a number and then call the metod *search()*. The loop must repeat until you enter 0.

An example to execute the program could be:

```
? 124
Found at index 3430834 : 23855452
? 1234
Found at index 616 : 63935
? 123
Not found : 58309823
? 0
```

PROBLEM 3

In mathematics wee works with sets and you can think of a set as a container for objects and thus a collection. A set is characterized by the following properties:

- *contains()*, which tests whether an element is in the set
- *union()*, which returns the union of the current set and another set
- *intersection()*, which returns the intersection of the current set and another set
- *differense()*, which returns the set difference between the current set and another set
- *subset()*, which returns a subset of elements that satisfy a certain condition

You must write a generic class that represents a set with the above operations. Start with a new project, you can call *SetProgram*. The project must have a class with the above methods and the following methods:

- *add()*, that add an element to the set – if the element already exists in the set the operation should be ignored
- *remove()*, that remove an element from the set – if the element is not in the set the operation should be ignored

Finally, the class must implement the iterator pattern:

```
package setprogram;

import java.util.*;
/**
 * Class representing a set with the classical set operations.
 * The class's goal is the sole to illustrate a generic class, but it has little
 * practical use since the methods complexity is bad.
 */
public class Set<T> implements Iterable<T>
{

 private ArrayList<T> list = new ArrayList(); // to the elements

  /**
   * Creates an empty set
   */
 public Set() {}
```

```
/**
 * Creates a set that contains elements.
 * @param t The elements that the set must contains
 */
public Set(T … t) {}

/**
 * Adds an element to the set. If the element already exists in the set,
 * the operation is ignored.
 * @param e The element to be added
 */
public void add(T e) {}

/**
 * Remove an element from the set. If the element does not exist in the set,
 * the operation is ignored.
 * @param e The element to be removed
 */
public void remove(T e) {}

/**
 * Implements contains.
 * @param e The element to be tested
 * @return true, if the element is found
 */
public boolean contains(T e) {}

/**
 * Implements union
 * @param A The set to create an union with this set
 * @return The union of the current set and A
 */
public Set<T> union(Set<T> A) {}

/**
 * Implements intersection.
 * @param A The set to create an intersection with this set
 * @return The intersection of the current set and A
 */
public Set<T> intersection(Set<T> A) {}

/**
 * Implements set difference
 * @param A The set to create a set difference between this set and A
 * @return The set difference between this set and A
 */
public Set<T> differens(Set<T> A) {}
```

```
  /**
   * Implements subset.
   * @param ok Selector, indicating the elements to be included in the subset
   * @return The elements that meets the selector
   */
  public Set<T> subset(ISelect<T> ok) {}

  /**
   * Implements the iterator pattern.
   * @return Iterator, that iterates through all elements in the set
   */
  public Iterator<T> iterator() {}
}
```

Regarding the method *subset()*, you must transfer a method that determines which elements to include in the subset. This can be done by means of an interface:

```
package setprogram;
/**
  * Interface, that defines a simple selector for elements in a subset.
  */
```

```
public interface ISelect<T>
{
 public boolean select(T e);
}
```

When you have written the class, you should test it from the *main()* method.

Note that the purpose of the task is only to show an example of a generic class, but the class has no particular practical interest, since it is inefficient. A set as a collection type, in turn, has interest and I will in a later book show how to implements a set that has a better complexity.

7.4 BOUND PARAMETER TYPES

Sometimes you can for generic methods and classes be interested in putting restrictions on the parameter type, so it's not all classes that can be used.

Above I have shown a generic method *linSearch()*, which searches for an element in an array. The method implements linear search, which is an algorithm that searches for an element in an array by comparing with the array's elements from start to finish. If the array has n items, it means using n/2 comparisons in average. This method in turn requires nothing about the elements and only they can be compared with *equals()*. Therefore, the method could be applied to arrays of arbitrary type.

Another search method is called *binary search* and can be used if you know in advance that the array is sorted, for example in ascending order. The principle is that you start to compare with the middle element, and if it is the element to sought, you are finished. Otherwise, it examines whether the element being searched is greater than or less than the middle element. Is it bigger than, you know, because of the array is sorted that element, if it exists, must lie in the right half, or else it must be in the left part. You now repeat the procedure, but only on the half of the array. This means that you have halved the number of elements to be searched. The same will happen next time, and in the last step (if the element is not found previously) you get a subarray whose length is 0. If the array has n elements, and each iteration halves the length, the necessary number of comparisons is limited by (as you just should accept), but for large values of n, it is much better than the n / 2. The result of this little quick presentation of binary search is that it is a search method that is much better than linear search, but it is important to emphasize that it assumes that the array to be searched, is sorted.

It is relatively simple to implement binary search in Java. Below is a generic method, but it requires that you can compare the elements (whether to take the left half or right half), and the parameter type must therefore implements the interface *Comparable*:

```
public static <T extends Comparable<T>> int binSearch(T[] arr, T elem)
{
 for (int a = 0, b = arr.length - 1; a <= b; )
 {
   int m = (a + b) / 2;
   if (arr[m].equals(elem)) return m;
   if (arr[m].compareTo(elem) < 0) a = m + 1; else b = m - 1;
  }
 return -1;
}
```

You should note the syntax:

```
T extends Comparable<T>
```

Here, the word *extends* means the type T either must to inherit a class or implement an interface. In this case it is the interface *Comparable<T>*, which means that objects of the type T can be compared. If you try to apply the method to objects that do not directly or indirectly implements *Comparable<T>*, you gets a compiler error.

One say that the parameter type *T* is bound to the type *Comparable<T>*. It can of course also be used in the context of generic classes. Below is a class *Point* to represent a point of two coordinates. The coordinates type must be a number, and the wrapper classes to numbers all extends the abstract class *Number.* Then I can write a class that can be used by all wrapper classes:

```
package generic;
public class Point<T extends Number>
{
 private T x;
 private T y;

 public Point(T x, T y)
 {
  this.x = x;
  this.y = y;
 }
```

```
public T getX()
{
 return x;
}

public T getY()
{
 return y;
}

public String toString()
{
 return "( " + x + ", " + y + " )";
}

public double length(Point p)
{
 return Math.sqrt(sqr(x.doubleValue() - p.x.doubleValue()) +
                  sqr(y.doubleValue() - p.y.doubleValue()));
}
```

```
 private double sqr(double x)
 {
 return x * x;
 }
}
```

You must especially note the method *length()*, which returns the distance between two points. In order to implement the calculation, the method uses *doubleValue()* on all the coordinates, and it is possible, as the compiler know that the *T* is a *Number* and thus makes this method available. The compiler knows on the other hand not where the type is an *Integer*, a *Double* etc., But it is also not necessary. Consider the following test method, which shows that you can determine the distance between two points, where the one has *Integer* coordinates, and the other has *Float* coordinates:

```
private static void test08()
{
 Point<Integer> p1 = new Point(5, 7);
 Point<Float> p2 = new Point(1.42, 3.14);
 System.out.println(p1);
 System.out.println(p2);
 System.out.println(p2.length(p1));
}
```

It is possible to bind a parameter type to several types with the following syntax:

```
class TesClass<T extends Type1 & Type2 & Type3)
```

Here only one of the types can be a class, while the other must be interfaces, and the one that is a class, must be the first.

EXERCISE 10

Add the method *binSearch()* to the class *Utils* in your class library. Then you must write a program, that you can call *SearchProgram2*. The program should be identical to the program *SearchProgram1*, except two things

1. the program must use *binSearch()* instead of *linSearch()*
2. the method *createArray()* must not shuffle the elements

Test the program and see if you can observe a time difference relative to *SearchProgram1*.

7.5 GENERIC TYPES AND INHERITANCE

In most cases, the use of generic types are without major problems, but the compatibility of inheritances is not quite what you might expect. If, for example you have a method

```
void show(Number t) { … }
```

so you can immediately transfer an *Integer*, a *Long*, a *Float* etc. as an actual parameter to *show()*, since they all specifically is a *Number*. Consider, however, the following methods:

```
private static void test09()
{
 Point<Integer> p = new Point(1, 2);
 show(p);
}

private static void show(Point<Number> p)
{
 System.out.println(p);
}
```

If you tries to compile these methods, you get an error that says that *p* is not compatible with *Point<Number>*. It would perhaps be expected when an *Integer* is compatible with *Number* because *Integer* inherits *Number*, but *Point<Integer>* does not inherit the *Point<Number>* and therefore the types are not compatible.

For the sake of the following, it is necessary to look at *type inteference*. It deals with how the compiler treats the call of a method and the definitions of objects to determine the type of the arguments and thus determine whether the call is possible. In this context, the compiler attempts to determine the most specific type which can be used for all arguments. This has importance with generic methods and is best explained with an example:

```
private static void test10()
{
 Number t1 = value(new Integer(23), new Double(3.14));
 // Float t2 = value(new Integer(23), new Double(3.14));
}

private static <T> T value(T t1, T t2)
{
 return rand.nextBoolean() ? t1 : t2;
}
```

The example does not perform something interesting and must only show that it is legal code that can be compiled correctly. *value()* is a generic method with one type parameter. The method has two parameters of this type, and returns a random of these objects. The first statement in *test10()* is legal, since both *Integer* and *Double* inherits *Number*. They are therefore type inferente, and the compiler can compile the code and type parameter will at runtime be a *Number*. In turn, the last statement in *test10()* is not permitted, since *Float* is not type inferent with *Integer* or *Double*. Put a little different type inference is a matter that the compiler out of the context can see the type to be used for a type parameter. That's why you can call the generic method *linSearch()* as

```
System.out.println(Tools.linSearch(arr, 12));
```

instead of writing

```
System.out.println(Generic.<Integer>linSearch(arr, 11));
```

It is possible to use wild cards, which is a ? and indicates an unknown type. It can be used to write more general methods. Consider the following code:

```
private static void test11()
{
 print1(create(2, 3, 3.14, 1.42));
// print1(create(2, 3, 5, 7));
// print1(create(2.1, 3.2, 5.3, 7.4));
}

private static <T> ArrayList<T> create(T … values)
{
 ArrayList<T> list = new ArrayList();
 for (T t : values) list.add(t);
 return list;
}

private static void print1(ArrayList<Number> list)
{
 for (Number n : list) System.out.print(n + " ");
 System.out.println();
}
```

which can be translated and run, and it is probably not so very strange. When *create()* is called, there are 4 actual parameters that because of the auto boxing are converted into objects of type *Integer* and the type *Double*. Because of the type inferencen the method *create()* creates an *ArrayList* of objects of the type *Number* and thus an object of the type *ArrayList<Number>*. This object can be transferred as a parameter to the method *print1()*. If you in the method *test11()* removes the first comment, you gets a compiler error. *create()* is called again with 4 actual parameters, but this time they are boxed as objects of the type *Integer*. The method will return an object to the type *ArrayList<Integer>*, and as explained in the beginning of this section, it is not compatible with the parameter to *print1()*. The problem can be solved with a wildcard:

```
private static void print1(ArrayList<? extends Number> list)
{
 for (Number n : list) System.out.print(n + " ");
 System.out.println();
}
```

The method *print1()* is now called with a parameter of type *ArrayList<T>* where *T* inherits (or implements) *Number*. The type *Number* is called an *upper bound*.

The wildcards do not needs to be bounded, and you can write a method like the following:

```
private static void print2(ArrayList<?> list)
{
 for (Object obj : list) System.out.print(obj + " ");
 System.out.println();
}
```

It will accept any *ArrayList<T>* as a parameter. Unbounded wildcards has not so many uses but can be used if the method only perceive an element as an *Object*.

A wild card may also have a *lower bound*:

```
private static void print3(ArrayList<? super Integer> list)
{
 for (int i = 0; i < list.size(); ++i) System.out.print(list.get(i) + " ");
 System.out.println();
}
```

The syntax is simple enough and means that the method as a parameter can use any *ArrayList<T>* where the parameter type *T* is a super type of *Integer*. This means that *Integer* must inherit or implement *T*. As an example you can write:

```
private static void test13()
{
 print3(create(2, 3, 3.14, 1.42));
 print3(create(2, 3, 5, 7));
 ArrayList<Double> list = create(2.1, 3.2, 5.3, 7.4);
//   print3(list);
}
```

The first *create()* creates as mentioned an *ArrayList<Number>*, which may be used as a parameter to *print3()*, because *Integer* extends *Number*. The second *create()* creates an *ArrayList<Integer>* that can also be used as a parameter, but the third creates an *ArrayList<Double>*, which is not a supertype of *ArrayList<Integer>*. You must specifically note that if you changed the last line to

```
print3(create(2.1, 3.2, 5.3, 7.4));
```

then things would work. The compiler will see that the method print3() requires an *ArrayList<? super Integer>* and the parameters to *create()* are boxed to objects of the type *Double*, but because of the type of inferencen between *Double* and *Integer*, the compiler will create an *ArrayList<Number>*.

8 LAMBDA EXPRESSIONS

If you in Java wants to transfer a method as a parameter to another method, this is done by means of an interface. An interface is a type and can specifically be used as a parameter to a method. Before I show how, I would say a little more about anonymous classes, which as the name says is a class that has no name. I have already mentioned anonymous classes above in connection with the iterator pattern.

8.1 ANONYMOUS CLASSES

As an example I will use the following types that I have seen on in the book Java 1:

```
package lambda;

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

```
package lambda;

public abstract class BankNote implements Note
{
 private int value;

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

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

It is types that defines banknotes. In Java 1, I also defined specific classes for Danish banknotes, but here I will show how these classes can be defined as anonymous classes. As an example you can define a 50 kr. banknote as follows (where all the code are written on a single line):

```
Note n1 = new Note() { public int getValue() {return 50; }
 public void print() {
  System.out.println("50 kr., Sallingsundbroen og Skarpsalling-karret"); }};
```

An anonymous class is defined by an interface (possibly a class), and the syntax is to write the interface name followed by parentheses:

```
INote n1 = new INote() { … };
```

and in the following block you then write the code for the methods that the interface defines. The class of the object in question has no name, and an anonymous class must therefore always be defined as part of an expression.

Anonymous classes are suitable and can simplify the code in situations in which you only needs a single object of a particular type. If the job had anything to do with banknotes, it would hardly be the case and the above and the following examples are also intended merely to show the syntax. Anonymous classes is most justified in those cases where the interface is used to define a few or perhaps only one method. The advantage is of course that in a program – and especially a GUI program – you do not has to write a series of simple classes that do other than to implement a single method, but there is also a disadvantage, as the code easily becomes difficult to read and even more difficult to write correct. The last you can help a little by not writting the code on a single line:

```
Note n2 = new Note()
{
  public int getValue()
   {
    return 100;
  }

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

It is something more readable, but comparing it with the fact that an anonymous class always occur in an expression, and in most cases it will be an assignment as above or as a parameter to a method, it is clear that you fast ends out with code which is hard to read. It is exactly what the lambda expression should do better.

Below is another example of an anonymous class, but this time the class is defined on the basis of the abstract class *BankNote*:

```
Note n3 = new BankNote(200) { public void print() {
 System.out.println("200 kr., Knippelsbro og bælteplade fra Langstrup"); }};
```

The syntax is essentially the same, but the example should show that you can pass values to the constructor, and the anonymous class only needs to implement the abstract methods of the class *BankNote*. Below is a third example:

```
Note n4 = new BankNote(500) {
 public void print()
 {
  System.out.println(
    "500 kr., Dronning Alexandrines bro og bronzespanden fra Keldby");
 }
};
```

Below is a method that has *Note* objects as parameters:

```
private static void print(Note … notes)
{
 int sum = 0;
 for (Note n : notes)
 {
  sum += n.getValue();
  n.print();
 }
 System.out.println("Value: " + sum);
}
```

The method is simple and does nothing but print the objects (performs the method *print()*) and the sum of their values. The parameter type is defined by an interface and the method knowns what this interface tells, but the actual parameters must naturally be objects created on the basis of concrete classes. Below is an example of calling the method, and you will primarily notice that one of the parameters are defined as an object of an anonymous class written directly in the call of the method:

```
print(n1, n2, n3, n4, new BankNote(1000) { public void print() {
 System.out.println("1000 kr., Storebæltsbroen og Solvognen"); } } );
```

8.2 METHODS AS PARAMETERS

In other programming languages, such as C and C ++ you can transfer references to methods as parameters to another method. This can not be done directly in Java, and it is necessary to transfer a method encapsulated in an object. The objects in this way transfered as a parameter to a method is usually defined by an interface. Consider as an example, the following interface, which defines a method that has an *int* parameter and returns a boolean:

```
interface ISelector
{
 public boolean select(int t);
}
```

Below is a method that prints the elements of an *int* array, but only the elements where an *ISelector* object returns true:

```
private static void print(int[] arr, ISelector s)
{
 for (int t : arr) if (s.select(t)) System.out.print(t + " ");
 System.out.println();
}
```

It is thus an example of how to transfer a method *select()* as a parameter to another method. Suppose there are defined the following array:

```
int[] arr = new int[120];
for (int i = 0; i < arr.length; ++i) arr[i] = i;
```

and assume that you want to use the above *print()* method to print all 2-digit numbers. You must then send an *ISelector* object as a parameter, and to create such an object, you must have a class:

```
class Select2 implements ISelector
{
 public boolean select(int t)
 {
  return t > 9 && t < 100;
 }
}
```

You can then print the numbers as follows:

```
print(arr, new Select2());
```

If instead you want to print all prime numbers, you can write the following class that implements the interface *ISelector*:

```
class SelectPrimes implements ISelector
{
 public boolean select(int t)
 {
  if (t == 2 || t == 3 || t == 5 || t == 7) return true;
  if (t < 11 || t % 2 == 0) return false;
  for (int n = 3, m = (int)Math.sqrt(t) + 1; n <= m; n += 2) if (t % n == 0)
   return false;
  return true;
 }
}
```

and then you can print the primes with the following statement:

```
print(arr, new SelectPrimes());
```

It is clear that in the situation where you have to transfer a method as a parameter to another method it is obvious to use an anonymous class. If you again wish to print all 2-digit numbers, you can use the following statement:

```
print(arr, new ISelector() {
 public boolean select(int t) { return t > 9 && t < 100; } });
```

where the *ISelector* object this time is defined on the basis of an anonymous class. The class *Select2* is then unnecessary. Similarly, one can print the prime numbers in the following way:

```
print(arr, new ISelector() {
 public boolean select(int t)
 {
  if (t == 2 || t == 3 || t == 5 || t == 7) return true;
  if (t < 11 || t % 2 == 0) return false;
  for (int n = 3, m = (int)Math.sqrt(t) + 1; n <= m; n += 2) if (t % n == 0)
   return false;
  return true;
 }
 });
```

but here are the benefits are not as big as this leads to code that is hard to read. If you want to print the 2-digit numbers, you can also use the following syntax:

```
print(arr, t -> t > 9 && t < 100);
```

It is an example of a *lambda expression*, and even if it seems mysterious, it is both simple to write and read. In its simplest form, the syntax of a lambda expression is

```
e -> expression
```

where the expression typically depends on *e*, and determines a value. In this case the compiler knows the *print()* method and knows that it requires an *ISelector* object and the lambda expression must therefore act on an *int* and return a *boolean*. Specifically, what happens is that the compiler creates an anonymous *ISelector* object that implements the *select()* method on the basis of the lambda expression.

It is also possible to write the selection of prime numbers by means of a lambda expression:

```
print(arr, t -> {
   if (t == 2 || t == 3 || t == 5 || t == 7) return true;
   if (t < 11 || t % 2 == 0) return false;
   for (int n = 3, m = (int)Math.sqrt(t) + 1; n <= m; n += 2) if (t % n == 0)
    return false;
   return true;
});
```

but then you are back to a code that is hard to read, and in this case I prefer to write a method:

```
public static boolean isPrime(int t)
{
 if (t == 2 || t == 3 || t == 5 || t == 7) return true;
 if (t < 11 || t % 2 == 0) return false;
 for (int n = 3, m = (int)Math.sqrt(t) + 1; n <= m; n += 2) if (t % n == 0) return false;
 return true;
}
```

and then write:

```
print(arr, t -> isPrime(t));
```

8.3 EXAMPLES OF LAMBDA EXPRESSIONS

The following examples are intended to show examples of lambda expressions, including variations of the syntax. As an example, I use a collection of postal codes, a collection that I have shown in a previous example in this book, and the starting point is thus the class:

```
package lambda;

public class Post implements Comparable<Post>
{
 private String code; // the zip code
 private String city; // name og the town

 public Post(String code, String city)
 {
  this.code = code;
  this.city = city;
 }

 public String getCode()
 {
  return code;
 }

 public String getCity()
 {
  return city;
 }
 public void setCity(String city)
 {
  this.city = city;
 }

 public String toString()
 {
  return code + " " + city;
 }

 public boolean equals(Object obj)
 {
  if (obj == null) return false;
  if (getClass() == obj.getClass()) return ((Post)obj).getCode().equals(code);
  return false;
 }

 public int hashCode()
 {
  return code.hashCode();
 }

 public int compareTo(Post post)
 {
  return code.compareTo(post.getCode());
 }
}
```

The class has changed a bit, and you should especially note that the class overrides *equals()* where objects solely are compared on there zip code, and the class implements the interface *Comparable<Post>* so that the objects are arranged only by the zip codes.

The class *Postcodes* is a collection of Danish postal codes, and the class is in many ways also the same as before, where the objects are created based on the nested class *Data*. The class is expanded with a few methods, and the one iterator is removed. The class is as follows:

```
package lambda;

import java.util.*;

public class Postcodes implements Iterable<Post>
{
 private ArrayList<Post> list = new ArrayList();
```

```
public Postcodes()
{
 Data data = new Data();
 for (int i = 0; i < data.length(); ++i)
  list.add(new Post(data.getCode(i), data.getCity(i)));
}

public int length()
{
 return list.size();
}

public Post get(int n)
{
 return list.get(n);
}

public Post get(String code) throws Exception
{
 for (Post p : list) if (p.getCode().equals(p)) return p;
 throw new Exception("Zip code " + code + " not found");
}
```

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

 public static class Data
 {
  …
 }
}
```

The program defines the following object, that I will use in the following:

```
private static Postcodes post = new Postcodes();
```

I will start with a search method that can search in the collection *post*. The method should have two parameters, where the one is the zip codes, while the other is the search criteria, which is a method that can select the objects to be included. Such a method could be defined with an interface in the following way:

```
public interface SearchPostcodes
{
 public boolean select(Post p);
}
```

and thus in the same manner as I before defined *ISelector*. However, it is unnecessary for the Java API defines a generic interface *Predicate<T>* for this purpose and method of searching can then be defined as follows:

```
private static ArrayList<Post> search(Postcodes post, Predicate<Post> select)
{
 ArrayList<Post> list = new ArrayList();
 for (Post p : post) if (select.test(p)) list.add(p);
 return list;
}
```

The predicate is called *select*, and it has a method called *test()* and parameter of the type *Post*. This method returns *true* if a zip code meet the criterion.

If, for example you want to find all the zip codes where the city name starts with "Ski", you can write:

```
private static void test03()
{
 print(search(post, p -> p.getCity().startsWith("Ski")));
}

private static void print(ArrayList<Post> list)
{
 for (Post p : list) System.out.println(p);
}
```

As another example, the following method finds all zip codes that start with 5 and where the city name includes the text "køb", but so that there is no distinction between uppercase and lowercase:

```
private static void test04()
{
 print(search(post,
  p -> p.getCode().startsWith("5") && p.getCity().toLowerCase().contains("køb")));
}
```

The above search method returns an *ArrayList<Post>* and in relation to lambda expressions the important parameter is *Predicate<Post>*. However, there are other options, and the following search method has also a *Consumer<Post>* as a parameter:

```
private static ArrayList<Post> search(Postcodes post, Predicate<Post> tester,
   Consumer<Post> action)
 {
  ArrayList<Post> list = new ArrayList();
  for (Post p : post) if (tester.test(p))
  {
    action.accept(p);
    list.add(p);
  }
  return list;
}
```

A *Consumer<Post>* object has a method called *accept()*, that has a *Post* object as a parameter. The idea is that you then can modify the objects before they are added to the list. As an example, chooses the method below that selects all zip codes where the city name states with "Ski", but before the objects are placed in the list, the city name is converted to uppercase:

```
private static void test05()
{
 print(search(post, p -> p.getCity().startsWith("Ski"),
  p -> p.setCity(p.getCity().toUpperCase())));
}
```

The last search method had a parameter of type *Consumer<Post>*, which has a *void* method *accept()*, which has a *Post* object as a parameter.

There is a generic interface *Function*, which is parameterized with the two types, that defines a method *apply()*, and wherein the two parameters denotes respectively the type of a parameter to *apply()* while the other is the type of a return value. Below is another search function, there as parameters has a *Predicate<Post>* and a *Function<Post, Post>*. The method has a *Post* object as parameter, and returns the same object again, but after the method *apply()* is used on the object:

```
private static ArrayList<Post> search1(Postcodes
post, Predicate<Post> tester,
  Function<Post, Post> func)
{
 ArrayList<Post> list = new ArrayList();
 for (Post p : post) if (tester.test(p)) list.add(func.apply(p));
 return list;
}
```

You should note that this time I have called the method *search1()* and not the *search()*. The reason is that otherwise the compiler can not distinguish between this search method and the previous, as there will not be overloaded on the return value of a method. Below is an example which uses the above search method:

```
private static void test06()
{
 print(search1(post, p -> p.getCity().startsWith("Ski"),
  p -> new Post(p.getCode(), p.getCity().toUpperCase())));
}
```

The difference is that this test method returns other objects than the previous test method that modifieds the original collection.

All of these options can be combined:

```
private static <X, Y> ArrayList<Y> search(Iterable<X> source, Predicate<X> tester,
  Function <X, Y> mapper, Consumer<Y> action)
{
 ArrayList<Y> list = new ArrayList();
 for (X x : source)
  if (tester.test(x))
  {
    Y y = mapper.apply(x);
    action.accept(y);
    list.add(y);
   }
 return list;
}
```

It is a generic method, which is parameterized with two types *X* and *Y*, and it returns an *ArrayList<Y>*. The method can be applied to any collection of objects of the type *X* that implements the iterator pattern. For the objects that are selected by the predicate, the method creates a new object of the type *Y*, and it is treated with the *Consumer<Y>* object before it is saved in the result. As an example, the following method selects the zip codes where the code begins with "78". These objects are returned as a string, but before the city name added to a StringBuilder:

```
private static void test07()
{
 StringBuilder builder = new StringBuilder();
 Collection<String> collection = search(post, p -> p.getCode().startsWith("78"),
  p -> p.toString(), s -> builder.append(s + ", "));
 for (String s : collection) System.out.println(s);
 System.out.println(builder.toString());
}
```

8.4 JAVA FUNCTIONAL INTERFACES

In the section above I have used the interfaces

- *Predicate<T>*
- *Consumer<T>*
- *Function<T, R>*

that are generic interfaces, all of which defines a single method, and they are actually defined to be used by a lambda expression. They are defined in *java.util.function*, and there are defined some more functional interfaces (next 50) and you are encouraged to investigate what interfaces are defined.

The *functional interfaces* looks like each other, and each interface has a single abstract method, called the *functional method* for that functional interface. The interfaces are as mentioned defined to be used with lambda expressions, but they are general purpose interfaces, and are available to be used by user code anywhere. Of course they do not identify all possible method prototypes to which lambda expressions might be used, but you should be aware that other Java packages also define functional interfaces.

8.5 EVENT HANDLERS

Probably the most important use of lambda expressions are attaching event handlers for components in a GUI program, and in fact I have already used lambda expressions in GUI programs several times. I have added a class to this project, called *Window*, and it opens the same window as in the first example in the book Java 2. The only change is that the event handlers are defined in a different way. The window has two buttons. One that erases the contents of a list box, and the only thing that must happen is that the model must be cleared. It can be written directly, using a lambda expression:

```
cmdClr.addActionListener(e -> model.clear());
```

The notation is appropriate if the event handler simply consists of a single statement or perhaps several quite simple statements. If the event handler is more complex, it is recommended to write a method:

```
private void addName(ActionEvent e)
{
 String name = txtName.getText().trim();
 if (name.length() > 0)
 {
  model.addElement(name);
  txtName.setText("");
  txtName.requestFocus();
 }
}
```

It must be a *void* method, and it must have a parameter of the type *ActionEvent*, but otherwise it is a rather straightforward method. You can then write the event handler as

```
cmdAdd.addActionListener(e -> addName(e));
```

Java, however, has a different syntax for the same thing:

```
cmdAdd.addActionListener(this::addName);
```

There are no benefits of the last writing, and one can even say that it is yet another use of the word *this*, but once you've used it for a while, the syntax is sensible enough.

9 COLLECTION CLASSES

A collection class is a container for objects, and a good example is the type *ArrayList*, which is discussed and used many times in the previous books. A collection class is a little more than just a container for objects since the class also provides a number of methods available which are used to manipulate the container's objects. Java has many other collection classes than *ArrayList*, and they differ in terms of how they are implemented and store their objects, as well as the services they provides in the form of methods. Overall reflects these collection classes the tasks that typically occur in programs to manipulate a family of objects.

It should immediately be said that the goal of this chapter is to give an overview of the collection classes, which classes exist and what they can be used for, but it is not a detailed review of the collection classes and how they work. However, it is the subject of the book Java 17.

Java collection classes are contained in a framework, which basically consists of three things:

1. abstract classes and interfaces that defines the various collection classes and their main characteristics
2. concrete classes that implements the interfaces in the framework
3. algorithms that are methods for a number of typical tasks such as searching and sorting

The whole framwork is generic in terms of both interfaces and classes, and the algorithms are defined as generic methods. The description of this framework is the goal of the following chapter. The first task is of course to get an overview of the content and in related to thatt how the various classes can be used. Moreover, it is necessary to have a general knowledge of how each class is implemented and working as it is a prerequisite for choosing the right class for the right task.

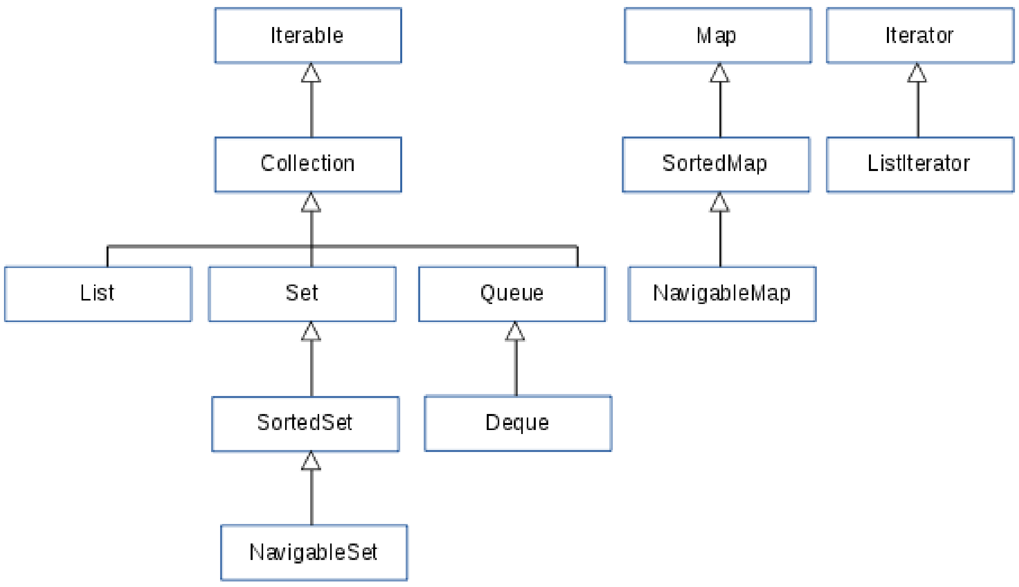
9.1 OVERVIEW OF THE COLLECTION CLASSES

There are many interfaces and abstract classes, which together define the characteristics of the collection classes, and I will mention below the main. Basic the framework is a hierarchy, but there are individual classes which are outside the hierarchy. The main are abstract classes and interfaces where I actually already have used some of them:

- *Iterable* is an interface that defines that a class implements the iterator pattern, and you can itererates the container's objects with a *for each* statement.

- *Collection* is a basic interface as the most of all collection classes in the hierarchy implements. It's rare that you directly refer to this interface, but it can be used if you write very general methods, and some of methods in the class *Collections* have parameters whose type is this interface.
- *List* is an interface that defines containers that have a sequence of objects, where each object is identified by an index. A *List* can not have holes, but it may contain the same objects more than once. An *ArrayList* is an example of a *List*.
- *Set*, *SortedSet*, *NavigableSet*, *Queue* and *Deque* are all interfaces for containers, not allowing the same element to exists several times. A *SortedSet* arranges the elements in sorted order, while a *NavigableSet* offers search options. *Queue* is an interface where you can access the oldest element and the class *Buffer* is actually a *Queue*. We call such a data structure for a FIFO structure for *Firts In First Out*. In contrast, a *Deque* represent a data structure where you always have access to the newest element, and you're talking about a LIFO data structure for *Last In First Out*. A stack is an example of a *Deque*.
- *Map*, *SortedMap*, *NavigableMap* are base classes for containers which stores key / value pairs. In a *SortedMap* the keys are stored in sorted order. A *NavigableMap* is a *Map*, which offer special search options. These three classes are not part of the same hierarchy as the other types of collection classes.
- *Iterator* and *ListIterator* are interfaces for iterators, and the difference is that the last interface defines the methods of traversing a collection in both directions.

The relationship between these types are as shown below:



I will not describe the individual types and the methods that they defines, but I will instead refer to the Java documentation, but it is all types that you must know and including their main characteristics.

The main konkete classes are as follows:

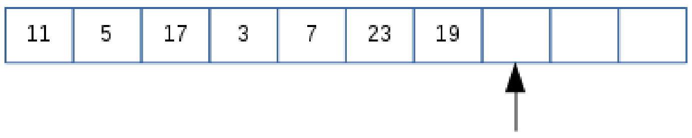
- *ArrayList.* It is a class I have used many times and can short be characterized as a dynamic array. The class implements the interface *List*. It is definitely the most frequently used of the collection classes.
- *LinkedList.* It is also a *List*, but is implemented in a different way than *ArrayList* and is an example of a double-linked list. In addition the class implements both *List, Queue* and *Deque*.
- *HashSet.* It is a *Set* and is implemented by hashing (explained below), and its primary quality is, that it is highly effective in terms of insertion and references to the elements.
- *TreeSet.* It is also a set, but is implemented in a completely different way by using a red-black tree. Unlike a *HashSet* a *TreeSet* guarantees that the elements are sorted.
- *HashMap.* It is a map, which is a data structure comprising of key/value pairs. It implements the data structure using hashing.

- *TreeMap*. It is a *SortedMap*, and in contrast to a *HashMap* does it guarantee that the elements are sorted by keys. The data structure is implemented by using a red-black tree.
- *PriorityQueue*. It is a data structure that corresponds to a *Queue*, but where one can insert elements after a priority. The data structure is implemented using a heap (see below).

In the following I will deal with these 7 data structures, focusing on their main features and show examples of how they are used. I also roughly outlines how they are implemented, as is necessary in order to use these classes effectively.

9.2 ARRAYLIST

An *ArrayList* is as mentioned above, a dynamical array, that automatically grows with the number of elements and you should have the following picture of an *ArrayList*:



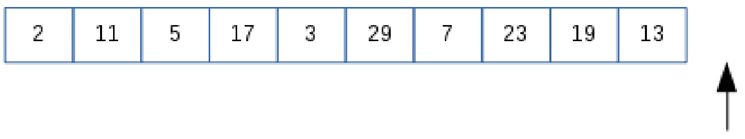
where there are seven elements in the list. When you create an *ArrayList* the internal array is empty, but when you add an item to the list, there is allocated room for 10 elements (the number 10 actually depends on the implementation). The arrow indicates where the next element is to be inserted, and the figure above illustrates such an *ArrayList* after that the following statements are performed:

```
ArrayList<Integer> list = new ArrayList<>();
list.add(11);
list.add(5);
list.add(17);
list.add(3);
list.add(7);
list.add(23);
list.add(19);
```

If you now further performs the following statements:

```
list.add(0, 2);
list.add(5, 29);
list.add(13);
```

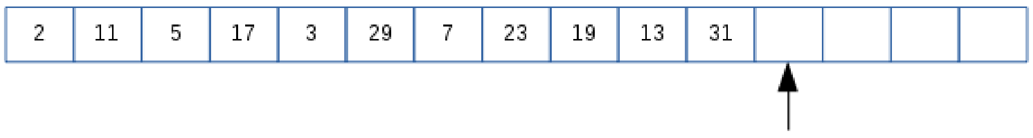
the picture is:



You should note that you can use *add()* with an index and then all elements to the right of the index most be moved to make room for the new element. You should also note that after these three statements, the entire capacity is used up, and the question is what happens if you add another element

```
list.add(31);
```

As mentioned previously, the idea of an *ArrayList* is that it automatically expands. When this happens the class uses an algorithm, that multiply the capacity by a constant (often it is 2), but in my present implementation, it is 1½. Therefore, the capacity after adding another element is 15:



The next time the array is extended, it will be to the capacity 22, then 33, then 49, then 73 and so on. The idea is that in most cases the addition of an element is extremely effective, and only in a few cases in which the array is to be extended, there is for a long task. As the array expands with increasing capacity, it becomes rarer that it must be expanded, and the result of all this is that on average, it is very effective to add elements to an *ArrayList*.

As a programmer you should not think so much of it, because it is something that happens internally, but you should know that the methods

- *add()*, that add an element to the end of the list
- *get()*, that returns the element at a particular index

are highly effective, while methods like

- *add()*, that adds an element at a particular index
- *remove()*, that removes an element

are less effective as they possibly must move many of the elements in the list (a list must not have holes).

In general, an *ArrayList* is highly effective and that it periodically must be expanded to increase the capacity is not something you have to think about. As an example is shown a list initialized with random numbers and then the method determines the sum of the numbers:

```
private static void test01(List<Integer> list, int n)
{
 long t1 = Calendar.getInstance().getTimeInMillis();
 while (n-- > 0) list.add(rand.nextInt());
 long t2 = Calendar.getInstance().getTimeInMillis();
 long s = 0;
 long t3 = Calendar.getInstance().getTimeInMillis();
 for (Integer t : list) s += t;
 long t4 = Calendar.getInstance().getTimeInMillis();
 System.out.println(t2 - t1);
 System.out.println(t4 - t3);
}
```

If I performs the method on my machine with the following statement

```
test01(new ArrayList<Integer>(), 1000000);
```

the method must be added a million elements to the list, and on my machine the result was:

```
31
6
```

This means that it has taken 31 milliseconds of initializing the list and 6 milliseconds to determine the sum. If I change the initialization statement to

```
while (n-- > 0) list.add(0, rand.nextInt());
```

and performs the method again, I get the result:

```
45876
6
```

and it will say that it has now taken over a ¾ minute to insert the numbers in the list, and it's quite a difference. The reason is that the elements this time are inserted at the beginning of the list and all other elements in the list must therefore be moved one place to the right. The example illustrates the great the difference in terms of efficiency depending on how the elements are inserted and it is a difference that you in practice must be aware of.

You can also create an *ArrayList* from another collection, where I here use the class *Arrays*, which has a method that creates a *List*:

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(11, 5, 17, 3, 7));
```

In this case, the list capacity is the number of arguments that have been transferred and hence 5 in this case.

As another example, regarding lists are shown a method that prints a collection:

```
private static void print(Collection<?> c)
{
 for (Object e : c) System.out.print(e + " ");
 System.out.println();
}
```

The method is trivial, but you should note two things. The parameter is a *Collection* parameterized with an unbound wildcard, and you can then transfer any *Collection* as a parameter. In addition, please note that the *Collection* inherits *Iterable* and therefore one can iterate over a collection with *for each*.

As a final note to the class *ArrayList* please note that it implements the interface *List* and thus *Collection* and *Iterable*. Moreover the class also implements three other interfaces: *Serializable*, *Cloneable* and *RandomAccess*. The first interface is discussed ealier, *Cloneable* I also mentioned earlier, and the last interface is a simple interface with no methods (such an interface is called a *marker interface*). It tells the runtime system that the elements in an *ArrayList* can be referenced by an index.

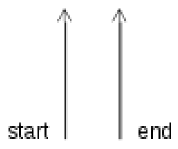
9.3 LINKEDLIST

It's also a list, and the class implements in the same way as an *ArrayList* the interface *List*. In addition, it implements the interfaces *Deque* and *Queue*. Finally, it implements *Serializable* and *Cloneable* but not *RandomAccess*. The class also has other methods than an ArrayList, but what matters is that it is implemented in a whole different way, as a double linked list.

If you creates a *LinkedList*

```
LinkedList<Integer> list = new LinkedList();
```

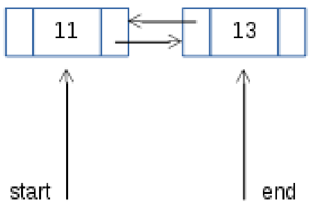
you have an empty list, which you should think about in the following way:



that is two pointers pointing respectively to start and end of the list. That the list is empty means that the two pointers both are *null*. The class has an *add()* method that works the same way as in an *ArrayList* and adds an element to the end of the list. If you perform the following statements

```
list.add(11);
list.add(13);
```

the picture of the list is as below:

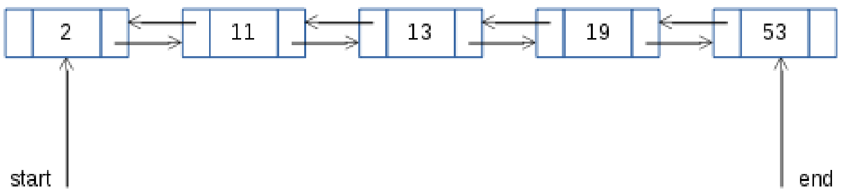


A *LinkedList* consists of so-called *nodes*, where a node includes a data item and a pointer to the previous node, and a pointer to the next node in the list. The first node (data element 11) has no predecessor, and its pointer to the previous node is *null*, and, similarly, the pointer to the next item in the node with the data element 13 is also *null*. In turn, node 11 points forward on node 13, while node 13 points back to node 11.

Compared to an *ArrayList* a *LinkedList* has several methods to add items to the list:

```
list.addFirst(2);
list.addLast(53);
list.add(3, 19);
```

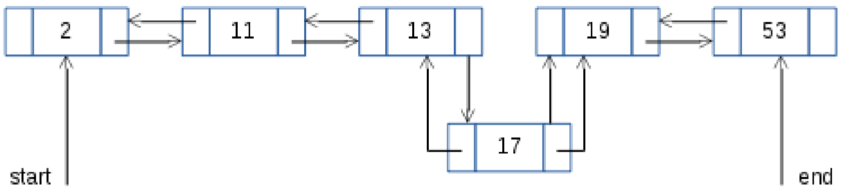
Here are *addFirst()* and *addLast()* are just special names for *add()* with index 0 and *add()* with no index. If the above statements are performed, the result is:



The idea with a *LinkedList* is that, when compared with an *ArrayList* it is simple to insert an element and delete an element in the middle of the list. If, for example you performs the statement

```
list.add(3, 17);
```

element 17 has to be inserted between 13 and 19:

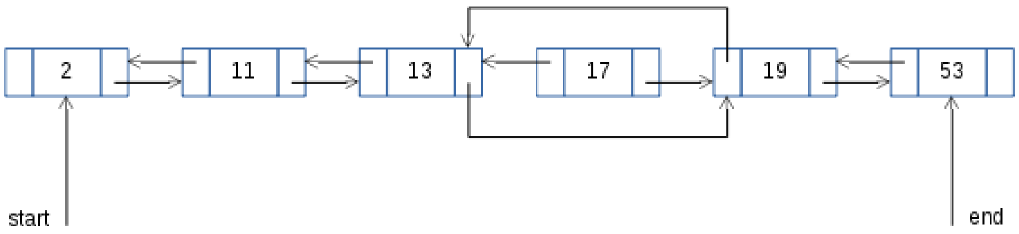


This means to create a new node, and then change the 4 pointers, but you should not in the same way as with an *ArrayList* move all elements to the right of the place where the new element should be inserted.

If you assume that the element 17 is inserted and you performs the statement

```
list.remove(3);
```

is the result

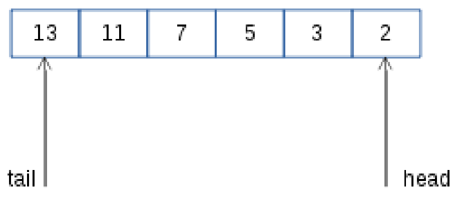


This means that there are changed two pointers, and there is thus no longer references to the item 17, which are then removed by the garbage collector. Again, the benefit is the same that deletion requires few operations. Both insertion and deletion of elements is simple, but requires special implementations of both inserting or deleting of elements in the ends of the list.

Another property of a *LinkedList* is, that it is not in the same way as an *ArrayList* allocates available capacity and periodically needs to expand, but on the other hand, each node uses two additional pointers to neighbor elements, which also fills, and it in reality means that a *LinkedList* not use less memory space than an *ArrayList* – and actually a bit more. With regard og inserting and deleting elements in the middle of the list you should be aware that although the operations are effective similar to that they simply consists of moving a few pointers, but you should be aware that the access to the list is only possible through the two pointers *start* and *end*, and there is therefore in principle necessary first to find the place where to insert or delete an element – the class does not implement the interface *RandomAccess*.

The most important characteristics of a *LinkedList* is that it is very effective to insert and delete items in the beginning of the list, and the like at the end of the list (as an *ArrayList*). These are the characteristics which are required by the implementation of a queue and a stack.

A queue is a data structure where you primary can add an item and removing an item but such that the element that is removed, always is the oldest element – it's the element been the longest in the queue. Typically, you have the following picture of a queue



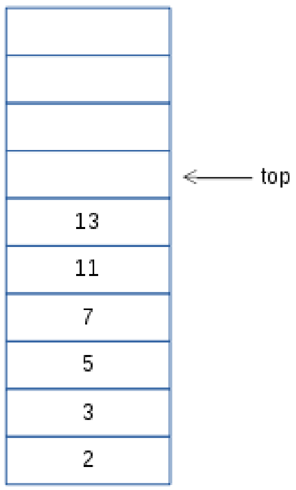
which adds elements where tail is pointing and remove the element where head is pointing. Since a *LinkedList* just performs well when inserting elements at the end of the list and to remove elements at the start of the list, it is suitable for implementing a queue, and that is expressed that the class implements the interface *Queue*. This interface defines 6 methods, that apart from the exceptions they can raise works in pairs like:

- *offer()* and *add()*, that add an element to the queue
- *poll()* and *remove()* that removes an element from the queue
- *peek()* and *element()*, that returns the oldest element without removing it

Below is a method that creates a queue and inserts 6 elements in the queue (similar to the figure above). Then the method prints the contents of the queue, and there after the queue is empy:

```
private static void test02()
{
 Queue<Integer> q = new LinkedList();
 q.offer(2);
 q.offer(3);
 q.offer(5);
 q.offer(7);
 q.offer(11);
 q.offer(13);
 print(q);
 while (q.size() > 0) System.out.print(q.poll() + " ");
 System.out.println();
}
```

A stack is look likes a queue, and it is again a data structure, where you can add and remove elements, but this time it is the element that was last added to the stack, that is is removed. A stack can be illustrated as shown here:



where there are 6 elements om the stack. The arrow indicates the place where the next element should be added. The implementation of a stack must be done using a data structure that is effective in terms of insertion of an element in the stack and then to remove the element again. These requirements meets both an *ArrayList* and a *LinkedList*. In the case of a stack the two operations for inserting elements in a stack and remove elements from the top of a stack is respectively called *push()* and *pop()*. A deque is a queue, where you can insert and remove elements both at the beginning and end of the queue, and when the class *LinkedList* implements the interface *Deque*, it is in principle a stack. The following method shows how you in Java can define and use a stack:

```
private static void test03()
{
 Deque<Integer> s = new LinkedList();
 s.push(2);
 s.push(3);
 s.push(5);
 s.push(7);
 s.push(11);
 s.push(13);
 print(s);
 while (s.size() > 0) System.out.print(s.pop() + " ");
 System.out.println();
}
```

There is also a second collection class called *ArrayDeque* which implements a deque by means of an *ArrayList*. It is with respect to a stack of slightly more effective.

EXERCISE 10

Write e program, that you can call *ListProgram*. The program should do the same as the method *test01()* above, that is add 1000000 random integers to a list, and then iterartes the list and determines the sum of the numbers. The program must for each of the two operations prints how long this will take, but this time the list must be a *LinkedList<Integer>*. Compare the result with the result af executing the method *test01()*.

PROBLEM 3

You should write a program called *StackProgram,* that can be used to test a *Deque*, when the type is used as a stack.

Add a simpel generic print method

```
private static <T> void print(T[] arr)
{
}
```

when the methed must print the array *arr* with all elements on the same line separated by a space.

It is possible to sort an array using two stacks, and the following algorithm can be used:

```
for each element t i in the array repeat
{
  as long t i less than the top of the left stack do
  {
   pop the left stack and push the element on the right stack
  }
  as long t is greater than the top of the right stack do
 {
   pop the right stack and push the element on the left stack
  }
  push t on the left stack
}
as long the left stack is not empty do
{
  pop the left stack and push the element on the right stack
}
loop over the array from start to end
{
  pop the right stack and insert the element in the array
}
```

Write a generic method *stackSort(T[] arr)* that implements the above algorithm and sorts an array.

Write a method *test1()*, that creates an array of the type *Integer* with 100 random numbers, such that every number is greater than 9 and less than 100. The method should print the array, sort it with the method *stackSort()* and print the array again.

Write a method *test(int n)*, that creates an *Integer* array with n random elementens. The method should sort the array with *stackSort()* and print how many milliseconds the sorting has taken. What happens if *n* is 100, 1000, 10000, 100000?

9.4 HASHSET

The interface *Set* defines a mathematical set and including the operations that you would typically expect to perform on sets. In chapter 6 you have written a class that could represent a set using an *ArrayList*, but I also noticed that the implementation of the set was not especially effective. Java, however, has a class *HashSet* which also represents a set, and is in turn effective. Consider the following method:

```
private static void test04()
{
 Set<Integer> A = new HashSet();
 Set<Integer> B = new HashSet();
 for (int i = 0; i < 10; ++i)
 {
  A.add(rand.nextInt(10));
  B.add(rand.nextInt(10));
 }
 Set<Integer> C = new HashSet(A);
 C.addAll(B);
 Set<Integer> D = new HashSet(A);
 D.retainAll(B);
 Set<Integer> E = new HashSet(A);
 E.removeAll(B);
 print(A);
 print(B);
 print(C);
 print(D);
 print(E);
}
```

This method creates two sets *A* and *B* and initializes them with digits. Next are determined respectively the union, intersection and set difference. If the method is performed the result could be:

```
0 1 5 6 8
0 1 2 4 7 8 9
0 1 2 4 5 6 7 8 9
0 1 8
5 6
```

The set *A* has 5 elements, and the set *B* has 7 elements. You should to note that the reason why the number of elements are different and there are not 10 elements in each set due to, that the same element can only occur once. You should also note that it looks like the elements are ordered, but it is only apparent, and in connection with a *HashSet* you must not assume any arrangement of the elements. You finally should notes the names of the methods, where *addAll()* is union, *retainlAll()* is intersection and *removeAll()* is set difference.

Consider as another example, the following method:

```
private static void test05(int n)
{
 Set<Integer> A = new HashSet();
 Set<Integer> B = new HashSet();
 long t1 = Calendar.getInstance().getTimeInMillis();
 for (int i = 0; i < n; ++i)
 {
  A.add(rand.nextInt(n));
  B.add(rand.nextInt(n));
 }
 long t2 = Calendar.getInstance().getTimeInMillis();
 System.out.println(A.size() + " " + B.size() + " " + (t2 - t1));
 long t3 = Calendar.getInstance().getTimeInMillis();
 Set<Integer> C = new HashSet(A);
 C.addAll(B);
 long t4 = Calendar.getInstance().getTimeInMillis();
 System.out.println(C.size() + " " + (t4 - t3));
 long t5 = Calendar.getInstance().getTimeInMillis();
 Set<Integer> D = new HashSet(A);
 D.retainAll(B);
 long t6 = Calendar.getInstance().getTimeInMillis();
 System.out.println(D.size() + " " + (t6 - t5));
 long t7 = Calendar.getInstance().getTimeInMillis();
 Set<Integer> E = new HashSet(A);
 E.removeAll(B);
 long t8 = Calendar.getInstance().getTimeInMillis();
 System.out.println(E.size() + " " + (t8 - t7));
}
```

This method creates two sets of random integers between 0 and n, and then creates the union, intersection and set difference. The method prints, how long each operation took: Below is the result of executing the method with n = 10000000:

```
6321515 6321068 9696
8646902 1365
3995681 3630
2325834 1184
```

It has taken about 10 seconds to try to create the two sets with 10 million members, and the result is sets at approximately 6 million elements. Secondly, it has taken about 1½ seconds to perform the union, about 3½ seconds to form the intersection and finally just about 1 second to form the set difference. The conclusion is that a *HashSet* is a very efficient data structure.

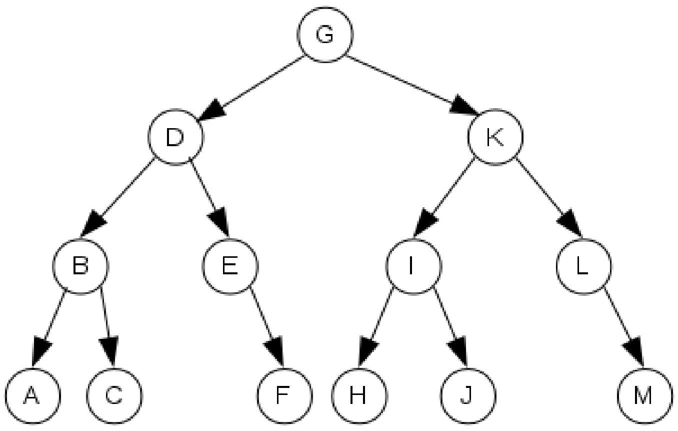
A *HashSet* is implemented by hashing, which is a technique in which an element in a collection can be found by a calculation. Internal, the collection is an array of sufficient size and the individual objects are placed in the array. When you add a new object, the class calculates using an algorithm where in the array the object must be placed, and it is placed there (if the place is not already used). Similarly, if you want to get a specific object in the collection, then the class just calculate the object's place and test whether it exists. There is direct access to the individual objects without any search, and that is exactly what makes the data structure extremely effective.

An *Object* has a method called *hashCode()*, which returns an *int*. When objects of a given type can be stored in a *HashSet*, the object's class must overrides the method *hashCode()* with value semantic. It is a requirement that two objects that are *equals()* also must have the same *hashCode()*, but it is also the only formal requirements. On the other hand, it is the programmer's responsibility to ensure that *hashCode()* is implemented in such a way that the method returns values that are uniformly distributed. It should be particularly noted that it is not a requirement that the two objects with the same *hashCode()* are *equals()*.

9.5 TREESET

The class *TreeSet* is also a *Set*, and you can exactly do the same as you can with a *HashSet*, and the difference is that a *TreeSet* arranges the elements in a specific order. The elements to be added to a *TreeSet* must therefore be objects created on basis of a class that implements the interface *Comparabel*. If so, it sounds reasonable to use a *TreeSet*, if you needs a *Set*, but you should be aware that a *TreeSet* not have the same effectiveness as a *HashSet*, and if you do not need that the elements are sorted, you should choose a *HashSet*. It's clear that it costs something when you have to insert elements and possible delete them again such that elements must be sorted.

It is not such that a *TreeSet* is inefficient and if you needs a sorted set, you should definitely not disregard a *TreeSet*. Internally a *TreeSet* does not consists of an array, but is instead a data structure composed of nodes that are linked together in much the same way as in a *LinkedList*. In a *TreeSet* elements are organized in a binary tree:



and the requirements are that each element may not have more than two direct sequels, and there should be exactly one element that has no predecessors and is called the root of the tree. Furthermore, it must be the case that if you stand in a particular node, then the left child must be *null* or less than where you stand, and the right child must be *null* or greater. The sum of all this is that if you stand in a certain node, then that node is root of a subtree in which all elements of the left subtree is less than the root and all elements in the right subtree is greater than the root. This is exactly what gives the tree and thereby also a *TreeSet* its qualities. You can then search the tree after the same principle as for binary search, and it is thus highly effective to ask whether an element is in a *TreeSet*, and iterating the data structure's elements such that the elements are visited in sorted order are equally effective.

EXERCISE 11

Create a program, that you can call *TreeProgram*. Add the following class to the project:

```
package treeprogram;
public class Person implements Comparable<Person>
{
 private String firstname;
 private String lastname;

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

 public String getFirstname()
 {
  return firstname;
 }

 public String getLastname()
 {
  return lastname;
 }

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

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

 public int hashCode()
 {
  return firstname.hashCode() ^ lastname.hashCode();
 }
 public int compareTo(Person pers)
 {
  if (lastname.equals(pers.lastname)) return firstname.compareTo(pers.firstname);
  return lastname.compareTo(pers.lastname);
 }
}
```

The class represents a person with a name, and there is nothing to explain, but you should note, that the class implements *equals()*, such that two *Person* objects are equals, if they have the same name. The class also implements *hashCode()*, and the method use a XOR operation, and although I have not yet explained what it is, it is enough to know that *hashCode()* is implemented, so the value is determined both by a person's first and last name. Finally you should note, that the class implements the interface *Comparable<Person>*, such that *Person* objects first are ordered after the last name and then by the first name.

Add the following method to the main class:

```
private static List<Person> createList()
{
 ArrayList<Person> list = new ArrayList();
 list.add(new Person("Karl", "Jensen"));
 list.add(new Person("Agnes", "Jensen"));
 list.add(new Person("Esben", "Hansen"));
 list.add(new Person("Abelone", "Andersen"));
 list.add(new Person("Knud", "Jensen"));
 list.add(new Person("Valborg", "Andesern"));
 list.add(new Person("Frede", "Jensen"));
 list.add(new Person("Olga", "Hansen"));
 list.add(new Person("Karlo", "Andersen"));
 list.add(new Person("Gudrun", "Hansen"));
 return list;
}
```

Then you should add a method, that prints the content of af collection, when the objects must be printed as a comma separated list on the same line:

```
private static void print(Collection<?> collection)
{
}
```

Write two methods, that from a list creates a *HashSet* and a *TreeSet*:

```
private static Set<Person> createTreeSet(List<Person> list)
{
}

private static Set<Person> createHashSet(List<Person> list)
{
}
```

Finally you should test the methods in the *main()* method:

```
public static void main(String[] args)
{
 List<Person> list = createList();
 print(list);
 Set<Person> set1 = createHashSet(list);
 print(set1);
 Set<Person> set2 = createTreeSet(list);
 print(set2);
}
```

9.6 HASHMAP AND TREEMAP

A *Map* is as mentioned a collection consisting of key/value pairs, where the key determines the value. An example of a concrete class is a *HashMap*, which is a generic class parameterized by two parameters, one for the key and the other for the value.

In the following example, the class *Person* is the same as in the previous exercise, and the same applies to the method *createList()*. The following method creates a *HashMap* where the key is a *String*, while the value is a *Person* object. The value of the key is the person's first name:

```
private static void test06()
{
 Map<String, Person> map = new HashMap();
 List<Person> list = createList();
 for (Person p : list) map.put(p.getFirstname(), p);
 for (String key : map.keySet()) System.out.println(key + "\t" + map.get(key));
}
```

Note how to add objects to a *HashMap* and you has to tell both the key and value. In this case it is assumed that all people have different first names. Finally, note how to traverse the data structure. First you get a *Set* with all keys, which you can traverse and using the key and the method *get()* you can determine the value associated with the key. If you performs the method, you get the result:

```
Knud        Knud Jensen
Karlo       Karlo Andersen
Agnes       Agnes Jensen
Olga        Olga Hansen
Karl        Karl Jensen
Frede       Frede Jensen
Abelone     Abelone Andersen
Valborg     Valborg Andesern
Gudrun      Gudrun Hansen
Esben       Esben Hansen
```

The name of a *HashMap* says that the keys internally are stored in a *HashSet*. There is another implementation of a *Map* called a *TreeMap*, and the name here is derived from, that the keys this time is stored in a *TreeSet*, which in turn means that if you traverse a *TreeMap*, the elements will be sorted by the key.

If you in the above method replaces the first statement with the following

```
Map<String, Person> map = new TreeMap();
```

and the method is executed the result is:

```
Abelone     Abelone Andersen
Agnes       Agnes Jensen
Esben       Esben Hansen
Frede       Frede Jensen
Gudrun      Gudrun Hansen
Karl        Karl Jensen
Karlo       Karlo Andersen
Knud        Knud Jensen
Olga        Olga Hansen
Valborg     Valborg Andesern
```

9.7 PRIORITYQUEUE

I have above mentioned a queue, and a priorityqueue is a queue, where the elements are added according to a priority, that means that the elements are orderede in the queue. The order can be defined by the natural ordering, that is the ordering defined by a class that implements *Comparable*, or by a *Comparator* provided at construction time. If a priority queue is relying on natural ordering, objects inserted in the queue must be defined by a class that implements *Comparable*. When elements are retrieved from the queue (for example with *poll()*), it is the *least* element with respect to the specified ordering that is retrieved.

The class is implemented by a heap, that is a data structure, which acts as a binary tree, but such that the element to be retrived next is a the root. It is an efficient data structure, and a *PriorityQueu* is then an efficient collection.

EXERCISE 12

In this exercise you should test the class *PriortyQueue*. Create a program, that you can call *PriorityProgram*. Add the class *Person* from exercise 11 to the project. Also take a copy of the method *createList()* in exercise 11 and paste the method to the main class.

Add a method *print()*

```
private static void print(Queue<Person> q)
{
}
```

when the method must remove all elements from the queue (*poll()* all elements) and print the elements as a comma separated list on a single line.

Next add a method *test1()*

```
private static void test1(List<Person> list)
{
}
```

that creates a *PriorityQueue<Person>* with natural ordering. The method must add all elements in the list to the queue, and then the method should call *print()* to print the queue.

In the *main()* method you must create a list (by *createList()*) and call *test1()*.

You should then write a method

```
private static void test2(List<Person> list)
{
}
```

that do exactly the same, but this time til ordering must be defined by a comparator, and the comparator should be defined by a lambda expression. The syntax to create a queue is:

```
Queue<Person> q = new PriorityQueue<Person>(list.size(),
 (Person p1, Person p2) -> p1.getLastname().compareTo(p2.getLastname()));
```

To use a comparator the constructor to the class *PriorityQueue* needs two parameters. The first parameter is an *int* and is an initial capacity. It has nothing to do with the comparator, and in principle you can use any positive number, but the idea with an initial capacity is to avoid the container to be expanded. The interface *Comparator<Person>* defines one method

```
int compare(Person p1, Person p2)
```

and to define this method as a lambda expression you needs an expression with two arguments. You should note, the use of the parentheses around the arguments. When you have written the method you must test it from the *main()* method.

Finally you should write a method

```
private static void test3(List<Person> list)
{
}
```

that do exactly the same as *test2()*, but this time til elements must be ordered by the first name.

If the program is executed, the result should be:

```
Abelone Andersen, Karlo Andersen, Valborg Andesen, Esben Hansen,
  Gudrun Hansen, Olga Hansen, Agnes Jensen, Frede Jensen, Karl Jensen,
  Knud Jensen
Abelone Andersen, Karlo Andersen, Valborg Andesen, Esben Hansen,
  Olga Hansen, Gudrun Hansen, Frede Jensen, Agnes Jensen, Knud Jensen,
  Karl Jensen
Abelone Andersen, Agnes Jensen, Esben Hansen, Frede Jensen, Gudrun Hansen,
  Karl Jensen, Karlo Andersen, Knud Jensen, Olga Hansen, Valborg Andesen
```

9.8 THE ALGORITMS

Java's collection API consist also of many methods, that can be used in practical programming and are methods that you often needs. They are members in the class *Collections,* and are all static methods whose first argument is the collection on which the operation is to be performed. Most of the methods operates on a *List*, but a few operates on a arbitrary *Collection*. The algorithms or methods are used for:

- sorting
- shuffling
- data manipulation
- searching
- composition
- finding extreme values

SORTING

The class *Collections* has a method *sort()*, that can be used to sort a *List*. It is a highly effective sorting method that use an algorithm called *merge sort*. There is two versions, where the one use the natural ordering, while the other use a comparator. The following method shows how to use *sort()* to sort *Person* objects in natural order:

```
private static void test07()
{
 List<Person> list1 = createList();
 List<Person> list2 = new LinkedList();
 for (Person p : list1) list2.add(p);
 Collections.sort(list1);
 print(list1);
 Collections.sort(list2);
 print(list2);
}
```

You should note the syntax (and simpler it may not be), and you should note that it also is possible to sort a *LinkedList*.

As another example the following method sorts a list with random integers and prints how many milliseconds the sorting has taken:

```
private static void test08(int n)
{
 List<Integer> list = new ArrayList();
 for (int i = 0; i < n; ++i) list.add(rand.nextInt());
 long t1 = Calendar.getInstance().getTimeInMillis();
 Collections.sort(list, (Integer a, Integer b) -> -a.compareTo(b));
 long t2 = Calendar.getInstance().getTimeInMillis();
 System.out.println(t2 – t1);
}
```

On my machine I get the result 3349 (about three seconds) to sort 10 million numbers.

SHUFFLING

There is also a method to shuffle a collection of objects. The method is called *shuffle()*, and the parameter is a *List<T>*. It is obviously not the most interesting method, but I have nevertheless in previous examples written my own shuffle method, and if you need, it's good to know that there is a method, which is just to apply.

```
private static void test09()
{
 List<Integer> list =
  new ArrayList(Arrays.asList( 2, 3, 5, 7, 11, 13, 17, 19, 23, 29));
 print(list);
 Collections.shuffle(list);
 print(list);
}
```

DATA MANIPULATION

The *Collections* class defines the following methods to manipulate the elements in a *List*:

- *reverse()*, that reverse the order of all elements in a *List*.
- *fill()*, that overrides all elements in a *List* with a specified value, and the method is usefull to initialize a *List*.
- *copy()* that copy all elements in one *List* to another *List*. The destination *List* must be at least as long as the source. If it is longer, the remaining elements in the destination *List* are unchanged.
- *swap()*, that swaps two elements in a *List*.
- *addAll()*, that adds all elements in an array to a *Collection*.

```
private static void test10()
{
 List<Integer> list1 = new ArrayList();
 Collections.addAll(list1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
 print(list1);
 Collections.fill(list1, 5);
 print(list1);
 List<Integer> list2 = new ArrayList();
 Collections.addAll(list2, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37);
 Collections.copy(list2, list1);
 print(list2);
}
```

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

SEARCHING

The class *Collections* implements also binary search:

```
<T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
<T> int binarySearch(List<? T> list, T key, Comparator<? super T> cmp)
```

As I have mentioned binary search, there is not more to say of it, but of course it is good to know that it already exists, so it is not necessary to implement it.

COMPOSITION

This group of methods contains two methods

- *frequency()*, that counts the number of times a specified element occurs in a *Collection*.
- *disjoint()*, that determines whether two *Collection* of objects are disjoint and then has no elements in common.

The following method shows to use this methods:

```
private static void test11()
{
 List<Integer> list1 = new ArrayList();
 Collections.addAll(list1, 11, 2, 3, 5, 7, 3, 11, 13, 17, 19, 3);
 System.out.println(Collections.frequency(list1, 2));
 System.out.println(Collections.frequency(list1, 3));
 System.out.println(Collections.frequency(list1, 11));
 System.out.println(Collections.frequency(list1, 12));
 List<Integer> list2 = new ArrayList();
 Collections.addAll(list2, 2, 4, 8, 16, 32, 64);
 List<Integer> list3 = new ArrayList();
 Collections.addAll(list2, 4, 8, 16, 32, 64);
 System.out.println(Collections.disjoint(list1, list2));
 System.out.println(Collections.disjoint(list1, list3));
}
```

```
1
3
2
0
false
true
```

FINDING EXTREME VALUES

That is two simple methods *min()* and the max() that, respectively, determines the minimum and maximum element in a *Collection()*. Both methods exists in two versions. The one takes only a *Collection* as parameter and returns the minimum (or maximum) element according to the elements' natural ordering. The other also has *Comparator* as a parameter.

EXERCISE 12

Write a program, that you can call *MinMax*. The *main()* method must create an *ArrayList* initialized with 10 random numbers of the type *Double*. After the list is created, it should be printed.

You must then use the methods *min()* and *max()* to prints, respectively, the smallest value and the biggest value in the list.

Finally you should du the samme, but this time the numbers must be compared with their numerical deviation from 0.5.

The result could be:

```
0.46732756489906613
0.6364287975820527
0.41794515249897457
0.4886474458162239
0.24707263700220072
0.30797749425175236
0.9921562698882159
0.23247661998977032
0.40622432064377734
0.5122684891300776

0.23247661998977032
0.9921562698882159
0.4886474458162239
0.9921562698882159
```

10 ANNOTATION

Annotations, are a form of metadata, and then data about a program that is not part of the program itself. Annotations have no direct effect on the execution of the program, but it is a way to write information to the compiler, that can use annotations to detect errors, but there are also annotations that are examined at runtime. Furthermore, there is some software tools that use annotations to generate code. Through the books you will see several uses of annotations but I will only show one in this book. Consider again the class *Person*, where I have only shown the part of the code that is related to annotations

```
public class Person implements Comparable<Person>
{
 …

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

 @Override
 public boolean equals(Object obj)
 {
  if (obj == null) return false;
  if (getClass() == obj.getClass())
  {
    Person pers = (Person)obj;
    return firstname.equals(pers.firstname) && lastname.equals(pers.lastname);
   }
  return false;
 }

 @Override
 public int hashCode()
 {
  return firstname.hashCode() ^ lastname.hashCode();
 }
 @Override
 public int compareTo(Person pers)
 {
  if (lastname.equals(pers.lastname)) return firstname.compareTo(pers.firstname);
  return lastname.compareTo(pers.lastname);
 }
}
```

In front of four of the methods I have written *@Override*. It is an annotation. It tells the compiler that the method is overridden from the base class or an interface, and the compiler will report an error if, for example, I have spelled the name of a method wrong. It remove a source of errors from the code, because you else may believe that a method is been overridden without it actually is. It is therefore recommended to use this annotation in front of all methods that are overridden.

11 PACKAGES

A Java program consists of many classes and interfaces. First, the classes that you yourself write, and also the classes that come from the Java API. To avoid name matches and to increase clarity, classes and other types are placed in packages. For example are Java's own classes divided into a very large number of packages. The basic package is called *java.lang*, and it contains all the basic classes such as *String*, *System*, *Object*, the wrapper classes etc. The classes in this package is immediately available for the runtime system, and you do not have to do anything special to use these classes. Things are different with classes in other packages. As an example, I have often used the class *Random*. This class is in the package *java.util*, which as the name says is a package of miscellaneous classes or tools. This means that the full name of the class is *java.util.Random*. If you have to create an object, you can therefore write:

```
java.util.Random rand = new java.util.Random();
```

When it is difficult to write (you has to write much), and it also makes the code harder to read, you can specify an *import* statement:

```
import java.util.Random;
```

which tells what the class *Random* means, and so you can just write the class name:

```
Random rand = new Random();
```

One should note that it is also the only thing that an *import* means and it is not a question that anything is imported into the program, but only that the class *Random* is directly available. When programs and classes usually uses many classes from the Java API, it can lead to many *import* statements, and therefore it is allowed to write

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

which shortly means that you can directly use all the classes and interfaces in the package *java.util*. It's something that is many opinions and arguments for and against, but I usually use the short notation.

When you in a NetBeans project creates a new class, it is automatically placed in a package with a *package* statement as the first statement:

```
package projectname;
```

The package name is by default the project name, but it need not be the case. A package statement must be the first statement in a source file (except for a possible comment), and the file must contains *import* statements as the next statements. A Java source file generally contains only one type and can contain only one *public* type, which must have the same name as the source file. If, for example you have the type

```
public class Str
{
```

the source file must be called *Str.java* and written with uppercase and lowercase letters as shown here. The file can contain several types, but it is generally advised not to do so, although there are sensible exceptions. When a file has several types, they can not be *public*, and therefore has package visability.

Until this place, my projects has only had one package, but if you write a large program with many classes, I often choses to place classes into multiple packages, so that classes on the same concept are in the same package.

12 FINAL EXAMPLE

In this chapter I will show the development of a simple GUI program. It's not a large program, and the goal is both to show the use of some of the concepts discussed in this book and to finish with an example where the focus again is on the process.

The process is described in relative detail, and the program is developed according to the MVC pattern. This pattern and in general the guidelines regarding the development of a program is first treated in the book Java 7, but you should easily be able to follow the process without having read Java 7. If in the description of the process is referred to concepts that you are not quite with on, just ignore it.

12.1 THE TASK

The task is to write a program that simulates a simple game, which consists of a square of 5 × 5 pieces arranged in random order. One of the pieces is empty, and you can move a piece by moving a neighbor to the empty piece. In the case below (the square on the left) you can then move the pieces H, E, L and J (you can not move diagonally). The game is solved when the pieces are arranged as shown in the example to the right.

K	G	V	M	R
I	S	X	U	F
C	O	A	E	T
Q	Y	H		L
N	D	P	J	B

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	X	Y	

You must assign a high score list to the program where to save the three best results. A result consists of the number of moves, used to solve the square, and it is therefore to solve the square by moving as few pieces as possible.

The square is basically a 5×5 square, but it must be possible to configure the program so that you can also play with a 3×3 square, a 7×7 square and a 9×9 square.

PROJECT START

I created a folder called puzzles, and it will contain all the files concerning this project. I have also created a subfolder called *doc*, which will contain all the project's documents. Preliminary contains this folder the above description of the task.

12.2 ANALYSIS

In this case, the analysis is simple, since the task is simple, and since the formulation of the task extensively describes the task. There are few issues to be clarified.

In the project description, the individual pieces are symbolized with letters. It's ok by a 5×5 square, but is not sufficient by respectively a 7×7 and 9×9 square. It is therefore decided to symbolize the pieces in a 3×3 and 5×5 square with letters, while there in a 7×7 and 9×9 must be used integers from 1 onwards.

With regard of the high score list it should be attached a high score list for each of the four levels of difficulty, as it does not make sense of comparing results from different levels of difficulty.

As the program's philosophy is entertainment and results can not be regarded as significant, it is agreed that high score lists are saved as regular files, and the program does not require database access. This means that anybody can delete the high score lists.

The program's idea is comparable with other entertainment programs like solitaire programs, which also is associated with a kind of high score list. It is therefore decided that the high score list should be developed general in order to be used in other applications. A high score will consist of

- a name (the user who has obtained the score)
- a score, that is an integer
- a time which indicates when the high score is achieved

Moreover, one must be able to specify an ordering that indicates where large values of the score is positive or negative. A high score list must have room for a certain number of high scores, as should be defined when the list is created.

REQUIREMENTS SPECIFICATION

The program should basically have the following functions:

1. *Play* where the user has an interaction with the program and rearrange the pieces until the square is resolved, or the user giving up.
2. *New game* where the user selects a new game. This means that the pieces are shuffled and the state of the program is set to start.
3. *Update high score list*, a feature which is triggered by the program when the square is resolved and the user's point indicates that the user should be on the list. The user must then enter an identification in the form of the name or an other text.
4. *Show high score list*, where the user chooses to view the high score list.
5. *Choosing the difficulty level* where the user must select a square size. This will automatically result in a new game.

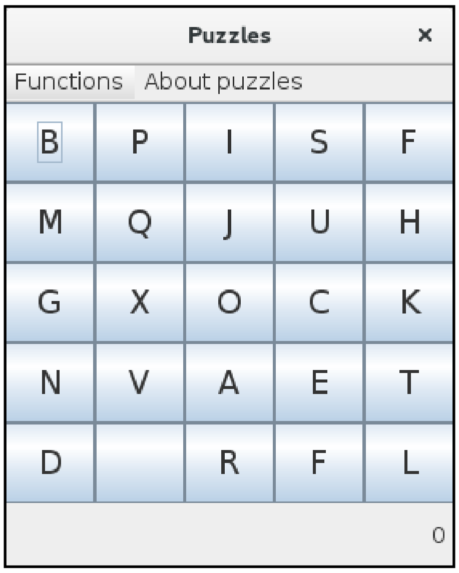
Regarding the high score list there are the following requirements:

1. The high Score list shoould show the top three and thus the players who have solved the square by moving as few pieces as possible.
2. Each high score must consists of a name, a date and time and the number of pieces that are moved.
3. It should not be possible (from the program) to delete the high score list.

The program's user interface shall consist of three windows:

1. The main window that appears when the application opens. The window displays 25 buttons corresponding to the puzzles pieces, a menu for selecting the above functions, and a status bar at the bottom of the window, showing how many pieces are moved.
2. A window that opens, if a player should be on the high score list (if there are not three players on the list, or the player's score is better than the last one on the list). The player should only be able to enter his name.
3. A window that shows the high score list.

In order to illustrate the user interface, a prototype has been developed called *Puzzles0*. It shows a design of the main window, and is designed as a grid consisting of 25 buttons. Furthermore, there is a menu for program's functions and a status bar to indicate how many pieces are moved. The prototype has no function, and nothing happens if you click on the pieces or select a menu item.



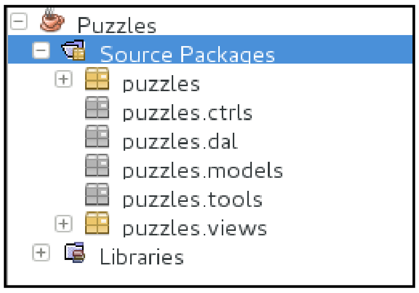
12.3 DESIGN

The following is both a description of the process, and hence how I have completed the design of the program *Puzzles* and partly it is a documentation of the program's user interface and classes.

ARCHITECTURE

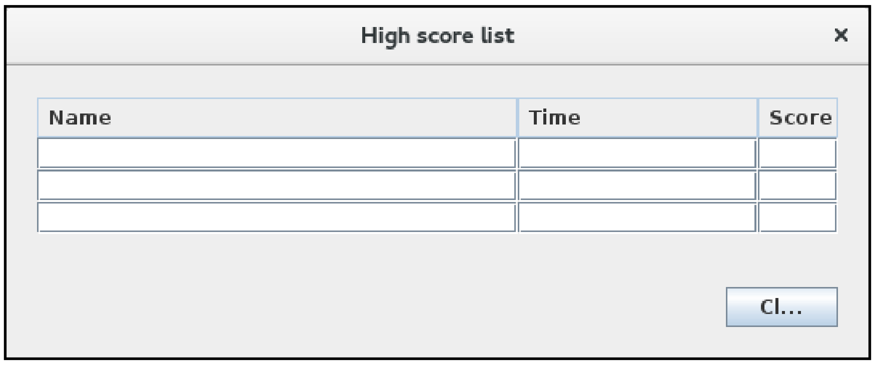
The starting point for the program's architecture is the MVC pattern, and to this is defined the following packages (se below), where:

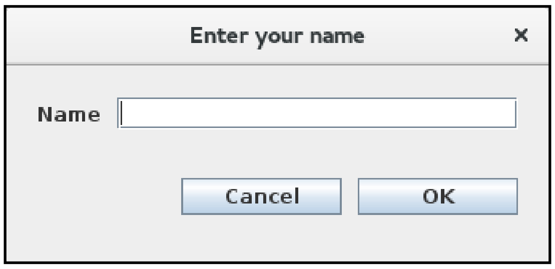
- *puzzles* that only contains the *main*-class
- *puzzles.ctrls* is for the controller classes
- *puzzles.dal* is the data access layer, that in this case should be used to classes for the high score list
- *puzzles.models* is for the model classes
- *puzzles.tools* is for helper classes
- brikker.*views* is for view classes and then the program's user interface



THE USER INTERFACE

The prototype from the analysis defines the most concerning the user interface and the main window is not changed in relation to it and belongs to the package *puzzles.views*. There are added two other windows respectively to show the high score list and for entering the name of a new high score. Are these windows opend from the main program, the results are:





In the same manner as the prototype there is not attached functions to this windows, and they should only show the user interface. They may thus be seen as an extension of the prototype.

To create these windows is added a class *GUI* to the package *puzzles.tools* containing static methods to create the user interface. These are examples of methods that could usefully be moved to a jar file with a class library, and maybe these tools when programming should be replaced with tools in the class library *PaLib*.

CLASSES

There is defined a single model class named *MainModel*. This class represents the game and keep track of its state. The class will also include the program logic and in accordance with the general guidelines for the architecture the class represents both the model and the controller for the main window.

There is one algorithm, which is not obvious. One can not simply shuffle the pieces randomly (using a random number generator) and if done, the square could not necessarily be solved. It is therefore necessary with a strategy (an algorithm) to shuffle the pieces. The idea is to start with a square that is solved, and then simulate random moves one of the pieces that can be moved (there are 2, 3 or 4 options). Repeats it many times, and you get a square where the pieces that are shuffled. The algorithm can be described as follows:

```
private void shuffle()
{
 // start with an ordered square
 // (r,c) refers to the blank piece - which to start is the last

 // loop over the desired number of substitutions (eg. 1000 or more)
 {
  // makes random one of the following operations
  {
   // go, if possible, a step up
   // go, if possible, a step down
   // go, if possible, a step to the left
   // go, if possible, a step to the right
  }
```

```
   // if you have reached a new position, swapped this piece of the blanks
   // and to start the next iteration based on the new position of the blank
  }
}
```

The high Score list is defined by two classes defined in the package *puzzles.dal*:

To solve the problem with the ordering of the high score objects the class *Highscore* is defined abstract and programs that use the high score list, therefore must define a derived class that implements *compareTo(Highscore hs)*. The two classes are shown below, where the comments are removed:

```
package puzzles.dal;

import java.util.*;
import java.io.*;
public abstract class Highscore implements
Comparable<Highscore>, Serializable
{
 private String name; // the name of the person who has achieved this score
 private int score; // the current score
 private Calendar time; // when this score is obtained

 public Highscore(String name, int score)
 {
  this.name = name;
  this.score = score;
  time = Calendar.getInstance();
 }
}

package puzzles.dal;

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

public class HighscoreList implements Iterable<Highscore>, Serializable
{
 private List<Highscore> list = new ArrayList();
 private int size;
 public HighscoreList(int size)
 {
  this.size = size;
 }

 public boolean addToList(int score)
 {
  throw new UnsupportedOperationException();
 }

 public boolean add(String name, int score)
 {
  throw new UnsupportedOperationException();
 }

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

The high score list has to be saved by ordinary object serialization. It provides a problem in terms of where the files should be saved. It has been decided that the files should be saved in a folder under */var/local/srv/data* where this folder must exist with write access.

12.4 PROGRAMMING

The programming includes writing the code for the application itself (see below). It has been decided not to use the class library *PaLib* and keep the class *GUI* with methods to the user interface. Similarly, it is left to the class *HighscoreList* to serialize and deserialie the objects. The finall classes (types) are as follows:

puzzles.tools

- *GUI*, which is a class with static helper methods to design of the user interface.

puzzles.views

- *MainWindow*, that represents the main window and concerning layout it is essentially unchanged from the prototype of the analysis. The class is expanded with the necessary event handling.
- *Options*, which defines constants for the user interface. In this case, it are only fonts.
- *ScoreDialog*, that is a dialog box for entering the name of a player to be on the high score list. The window has the same look as shown in the design, but the code is changed to ensure a more stable layout.

- *ScoresDialog*, there is a dialog box that shows the high score list. The window has the same look as shown in the design, but the code is changed to ensure a more stable layout.

`puzzles.ctrls`

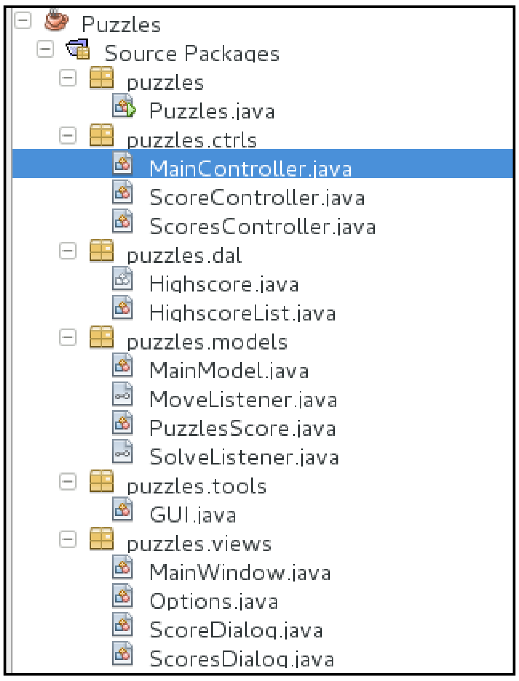
- *MainController*, which is a controller class to the main window. This class is in this case trivial and is most included to comply with the MVC.
- *ScoreController*, there are controller class for *ScoreDialog*. The class's constructor validates whether a user should be on the high score list and if so opens a *ScoreDialog*.
- *ScoresController*, that is controller class to *ScoresDialog*.

`puzzles.models`

- *MainModel*, that is the model for *MainWindow*. The class also implements the logic of the game.
- *PuzzlesScore*, that is extended from *Highscore* and is a concrete *Highscore* class.
- *MoveListener*, that is an interface, that defines an observer for changes in the state of the model.
- *SolveListener*, that is an interface, that defines an observer for solving the square.

`puzzles.dal`

- *Highscore*, that is an abstract class, that represents a high score. The class is abstract, because it defines but not implements the interfacet *Comparable<Highscore>*.
- *HighscoreList*, that defines a high score list and serialize and deserialize *Highscore* objects.



```
Puzzles
  Source Packages
    puzzles
      Puzzles.java
    puzzles.ctrls
      MainController.java
      ScoreController.java
      ScoresController.java
    puzzles.dal
      Highscore.java
      HighscoreList.java
    puzzles.models
      MainModel.java
      MoveListener.java
      PuzzlesScore.java
      SolveListener.java
    puzzles.tools
      GUI.java
    puzzles.views
      MainWindow.java
      Options.java
      ScoreDialog.java
      ScoresDialog.java
```

To test that the program works, the folowing must be validated:

- is the model created correct and are the pices shuffled sensibly
- are the pieces moved correctly
- can the square be solved and are the players added to the high score lists in the right order
- works the function *New game*
- is it possible to select a new difficulty level
- can the high score lists be shown
- shows *About Puzzles* the correct window
- is the program's look and feel at it should be

The first point is best solved by playing the game.

The next points can only be tested by using the program and thus play the game. It is relatively simple to test the 3×3 to 5×5 squares, but the 7×7 or 9×9, it is more difficult since it takes a long time to solve the square. You can solve this problem by changing the loop in the method *shuffle()* so that it only takes one or a few exchanges.

The last point can only be tested in collaboration with the user and get his agreement that the user interface is satisfactory.

12.5 TEST

When a program is finished, it must be tested in real environments and any of other than the programmer. In this case there is not much to do in the finally test, because the program does not have to communicate with other programs, but the program must of course be tested, and it must be in correct user environment.

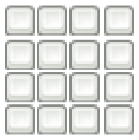
In this case, you can create a folder somewhere on your hard drive and then copy the program's jar file to that folder and start the program from a prompt. Then there is not much else to do than to play and try to solve the squares. Here you must pay particular attention to the high score list, and the players are correctly added to the lists. In addition, you must have an eye on the visual, and if all looks as it should do. Specifically, you can use the following test cases:

1. Set the level to a et 3×3 square
2. Solve the square and add the user to the high score list
3. Solve the square again, but this time with many swap, such that the user get a high score with many moves
4. Solve the square a third time
5. Show the high score list and examine whether the three players are correctly inserted
6. Solve squares until another player is added to the high score list
7. Display the high score list to ensure that the last player is properly inserted
8. Repeat the above with a 5×5 square
9. Manually delete the two files to high score lists
10. Display the high score lists for both a 3×3 and 5×5 square and ensure that they are empty

If these test cases are performed satisfactorily, I will consider the program as tested and ready to use.

12.6 THE LAST STEP

As the last part of the task that must be written a script that can install the software on a computer. As the first I've drawn an icon for the program



that I have called *puzzles.png*. Then I have created a directory named *puzzles* that contains the program's jar file and the above icon and the following script. To install the program you must then make the directory puzzles to the current directory and execute the script as

```
sudo ./puzzles.sh
```

This will create the necessary directories and the program is copied to */usr/local/games* and you get an icon on the desktop that refers to the program. The script is as follows:

```
#!/bin/bash
DIR=$(pwd)
cd /var/local
if [ -d srv ]
then
 cd srv
 if [ -d data ]
 then
 cd data
 else
 mkdir data
 fi
else
 mkdir srv
 cd srv
 mkdir data
 cd data
fi
if [ -d highscores ]
then
 rm highscores/* 2> /dev/null
else
 mkdir highscores
fi
cd "$DIR"
cp Puzzles.jar /usr/local/games
cp puzzles.png /usr/local/games
cd /usr/local/bin
echo 'java -jar /usr/local/games/Puzzles.jar' > puzzle
chmod 755 puzzle
cd /usr/share/applications
echo '[Desktop Entry]' > puzzles.desktop
echo 'Name=Puzzles' >> puzzles.desktop
echo 'Exec=puzzle' >> puzzles.desktop
echo 'Icon=/usr/local/games/puzzles.png' >> puzzles.desktop
echo 'Type=Application' >> puzzles.desktop
echo 'Categories=Games' >> puzzles.desktop
echo 'Encoding=UTF-8' >> puzzles.desktop
```

It is a relatively simple script, but short the following takes place on the condition that the current directory is the directory with the script, and the jar file and the icon:

1. the name of the directory is stored in a variable named *DIR*
2. the current directory is changed to */var/local*
3. here is created a subdirectory *srv*, if it does not already exist, and here again a subdirectory *data*
4. if *data* has a subdirectory *highscores* delete all files in this directory, and otherwise create a subdirectory *highscores*
5. the current directory is change to the directory stored in *DIR*
6. the jar file and the icon are copied to */usr/local/games*
7. current directory is changed to */usr/local/bin*
8. here is created a simple script (only one line) containing the start command
9. for this script the rights is set so that everyone has *read* og *execute*
10. the current directory is changed to */usr/share/applications*
11. here the script creates a file named *puzzles.desktop*, and it's the commands in that file that creates an icon for the program