Poul Klausen

JAVA 8

Multithreaded programs

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 8: MULTITHREADED PROGRAMS

SOFTWARE DEVELOPMENT

Java 8: Multithreaded programs: Software Development
1st edition

ISBN 978-87-403-1820-3
Peer review by Ove Thomsen, EA Dania

CONTENTS

1 FOREWORD

This book is the eighth 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 deals with threads, and how you in Java can synchronize threads that share resources. The use of threads is an important issue in programming, but it is also a technical area to which are attached many details. The book has therefore primarily focus on programming and the language Java, and only in the final example, there is again focusing on system development and thus the process. After reading the book the reader should be able to use threads in practice and have an understanding of where threads can be used in practical programming. The book requires knowledge of programming and Java corresponding to the content of the first four books in this series and in the interests of the final example is also assumed knowledge of the contents of the previous book, which deals with system development. The book ends with an appendix that gives a general introduction to the Java virtual machine JVM and what it is.

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

When you start a program it creates a thread, as compared to the runtime system is responsible for carrying out the program's statements. You can think of a thread as a part of the program code that has its own life and can be performed independent of the runtime system in competition with threads from other applications or a thread from the same program. In order for a thread to run and execute anything, it must have a processor to run on, and even though computers today usually have multiple processors, there will always be many more active threads than there are processors. A thread has only a processor in a very short time, after which it is interrupted by the system and put in a queue while another thread get the processor. Since all threads in this way with very short time intervals will have a processor available, it seems to us users as if all threads are running in parallel next to each other.

When you start a program so is created, as mentioned a thread to execute the program's instructions. This thread is called the *primary thread*. An application can, however, create multiple threads, and many programs do, and a GUI program will at least create two threads. The idea of allowing a program to create multiple threads is to achieve parallelism, so it for users seems as the program is doing several things simultaneously. For example a word processor can perform a spell checking while the user continues entering text. The subject of this book is to show how Java creates a thread and the challenges it leads to, but also to show what you can use threads in practice. Indeed, viewed from the programmer some challenges are attached to threads and the book contains many technical details.

1.1 CREATE A THREAD

I'll start by showing how a program can create threads, and the following program creates four threads beyond the primary thread:

```
package thread01;

import java.util.*;

public class Thread01
{
 public static void main(String[] args)
 {
  (new Thread(new Worker1())).start();
  new Worker2();
  (new Thread(new Runnable() {
   public void run() { ToDo.work(2, 5); } })).start();
  (new Thread(() -> ToDo.work(2, 5))).start();
```

```
   ToDo.work(5, 10);
  }
 }

 class Worker1 implements Runnable
 {
  public void run()
  {
   ToDo.work(2, 5);
  }
 }

 class Worker2 extends Thread
 {
  public Worker2()
  {
   start();
  }

  public void run()
  {
   ToDo.work(2, 5);
  }
 }

 class ToDo
 {
  private static Random rand = new Random();

  public static void work(int a, int b)
  {
   print("started");
   for (int i = 0, n = rand.nextInt(b - a) + a; i < n; ++i)
   {
    print("working.........");
    work();
   }
   print("terminated");
  }

  private static void print(String text)
  {
   long id = Thread.currentThread().getId();
   System.out.println("[" + id + "] " + text);
  }
```

```
 private static void work()
 {
  double y;
  for (int i = 0; i < 1000000L; ++i) y = Math.cos(Math.sqrt(rand.nextDouble()));
 }
}
```

I'll start with the class *ToDo*, which in principle do not have anything with threads to do. The class represents a work that the computer has to perform, and the important thing is that it is a work that takes time, and then strains the machine's processor. The private method *work()* performs works not used for anything but the work consists of performing one million calculation in which a calculation is to perform a mathematical function which uses many CPU instructions. The second version of the method *work()* (the *public* version) has two parameters, and call the private *work()* a number of times determined by these parameters. The method first prints a message on the screen, and after the work is completed, it prints another message. For the message you should note the statement

```
long id = Thread.currentThread().getId();
```

A thread is identified by an ID, which is an integer and is assigned when the thread is created. The primary thread has number 1, while custom threads are assigned a number from 10 onwards. When the method *work()* prints a message (the method *print()*), it prints also the current thread ID (the thread that performs the method *print()*).

In Java is a thread is represented by the class *Thread*. The class *Worker2* inherits the class *Thread* and represents as such a thread. The class *Thread* has a method called *run()* and it is the method that executes when the thread is started, and in this case it happens in the constructor of the class *Worker2*. It is the programmer of a *Thread* class that must override the method *run()* and determine what the thread has to perform and the thread runs until the method *run()* terminates. In this case, the thread performs the method *work()* in the class *ToDo*.

The class *Thread* implements the interface *Runnable*, which alone defines the method *run()*. Another way to define a thread is to transfer a *Runnable* object as a parameter to the *Thread* class's constructor and in this way tells what kind of a *run()* method to be executed. The class *Worker1* is a simple class that implements *Runnable* and thus must implement the method *run()*, and the class defines then *Runnable* objects. In this case, the method *run()* performs the same as in the thread class *Worker2*.

Back's is the *main()* method. The first statement creates a new *Thread* object and start a thread. The parameter to the constructor is a *Runnable* object, and in this case it is an object of the type *Worker1*. The next statement starts another thread, but this time by instantiating an object of the type *Worker2*. Here it is the constructor of the class *Worker2* that starts the thread. Then there is the third statement that does the same as the first statement and thus starts a thread that performs the method *work()*, but instead to create a *Runnable* object of the type *Worker1*, the statement instantiates a *Runnable* object using an anonymous class. The fourth statement does the same, but here it happens instead using a lambda expression. Back there is the last statement, which simply calls the *ToDo* class's *work()* method. That is, it is the primary thread that performs this statement, and the meaning is that you have to see that the primary thread is running in parallel with the other threads. If you executes the program, the result could be the following:

```
[11] started
[10] started
[10] works........
[12] started
[11] works........
[12] works........
[1] started
[13] started
[1] works........
[13] works........
[12] works........
[13] works........
[12] works........
[12] works........
[12] terminated
[10] works........
[11] works........
[1] works........
[11] terminated
[1] works........
[10] terminated
[13] works........
[1] works........
[13] terminated
[1] works........
[1] works........
[1] works........
[1] works........
[1] terminated
```

Here you should note that the 5 threads are running on shift. When the threads are carrying out work that takes time, they are periodically interrupted, and a second pending thread is running instead of. It is important to note that one can not assume anything about when a thread is interrupted, and when the thread again is started. It is managed by the operating system and is determined by the tasks to be performed on the machine.

1.2 THREADS PROPERTIES

I will then show an example that prints properties for a thread. The program starts two threads and shows examples of information regarding the threads state. The program also shows how to suspend a thread, which means that the thread does not have a CPU available before it is again ready.

```
package thread02;

public class Thread02
{
 public static void main(String[] args)
 {
  System.out.println(
   ("Number of processors: " + Runtime.getRuntime().availableProcessors()));
```

```
  Thread th1 = new Thread(new InfoThread(), "Thread number one");
  Thread th2 = new Thread(new InfoThread(), "Thread number two");
  System.out.println(th1.getState());
//   th1.setDaemon(true);
  th2.setDaemon(true);
  th1.start();
  th2.start();
  System.out.println(th1.isAlive());
 }

}

class InfoThread implements Runnable
{
 public void run()
 {
  Thread th = Thread.currentThread();
  System.out.println("[" + th.getId() + "] " + th.getName() + " is started");
  System.out.println(
   "[" + th.getId() + "] " + (th.isDaemon() ? "Deamon" : "None deamon"));
  System.out.println("[" + th.getId() + "] " + th.getState());
  try
  {
   Thread.sleep(2000);
  }
  catch (Exception ex)
  {
  }
  System.out.println("[" + th.getId() + "] " + th.getName() + " is terminated");
 }
}
```

The thread is defined by the class *InfoThread* that implements *Runnable* and thus the method *run()*. The method first prints the thread's id and its name. The class *Thread* has a method, so you can assign a thread a name, but you can also provide a thread a name when it is created. A thread may be a *daemon thread* or a *none daemon thread* (also referred to as a worker thread). A daemon thread is intended as a helper thread to another thread, and the difference is that a daemon thread is automatically terminated when the program's last none daemon thread terminates, and when the primary thread is a none-daemon thread, the daemon threads automatically terminates when the program terminates. In contrast, the primary thread (and therefore the program) does not terminate until all none daemon threads is completed. The next print statement in the *run()* method prints whether the current thread is a daemon thread. The third statement prints the thread's state, which can be

- NEW, a thread is created, but not yet started
- RUNNING, the thread is performed on a processor
- BLOCKED, a thread is blocked and is waiting on a lock
- WAITING, a thread waits on a notification from another thread
- TIMED_WAITING, a thread waits on a notification because of a timeout
- TERMINATED, the thread is terminated

After these three print statements are executed, the *run()* method performs a *sleep()*, which means that the thread is suspended for a certain period. The time is specified in milliseconds, and in this case the thread is suspended in 2 seconds. That a thread is suspended means that in the period in question it is not active and does not participate in the resource allocation. Especially it uses no CPU time. When the thread again becomes active and is running, it prints another message with its id and name, and then the thread exits.

The *main()* method starts by printing the number of processors that are available on the current computer. Next, it creates two threads, both threads defined with a *Runnable* object of type *InfoThread* as parameter. Note that the two threads at the same time are given a name. Next is printed the state of the first thread, which at that time will be NEW. Then is defined that the other thread should be a daemon thread. Finally the two threads are started, and the *main()* method prints that the first thread is "live". If you run the program, the result could be the following:

```
Number of processors: 8
NEW
true
[10] Thread number one is started
[11] Thread number two is started
[10] None deamon
[10] RUNNABLE
[11] Deamon
[11] RUNNABLE
[10] Thread number one is terminated
[11] Thread number two is terminated
```

Here you should specifically note that maybe the last thread not writes that it is completed. The reason is that it is a daemon thread and as soon as the first thread terminates, also the primary thread terminates and thus also the second thread. In the *main()* method is a comment in front of the statement that defines the first thread as a daemon thread. If you remove this comment, the primary thread does not wait for the two threads and finish with the same – and the two threads are immediately terminated without completing their work.

The program here shows some of the properties a thread can have and the services that the *Thread* class provides. There are other options such to change a thread's priority. You must, however, usually not do that, unless you have a good reason.

As another example of a program that starts multiple threads, you can consider the following program that starts 100 threads:

```
package thread03;

public class Thread03
{
 public static void main(String[] args)
 {
  for (int i = 0; i < 100; ++i) (new Thread(() -> work())).start();
 }
```

```
 private static void work()
 {
  print("Startet");
  double y = 0;
  for (int i = 0; i < 10000000L; ++i) y = Math.cos(Math.sqrt(2));
  print(String.format("%1.4f", y));
 }

 private static void print(String text)
 {
  System.out.println("[" + Thread.currentThread().getId() + "] " + text);
 }
}
```

Compared to the two first examples there are nothing new, but the program should show what happens if you start many threads, all of which are CPU intensive. There are started 100 threads, but what matters is that there are started many more threads than there are CPUs. If the program is performed, you must observe that all threads are started but prints the start message in unpredictable order. In turn, the threads performs their work until they quit, and again the order is unpredictable. It depends on when the operating system interrupts the threads and decides which threads should be started again.

EXERCISE 1

Write a program that creates and starts two threads. The one thread must perform a loop that iterates 10 times. For each iteration, the thread should print an integer between 0 and 99, after which it must be suspended for a random period less than one second. The second thread should act in the same way, but instead print a random decimal number. My solution is called *StartThreads*.

2 JOIN

Consider the following program:

```
package thread04;

public class Thread04
{
 private static long prime;

 public static void main(String[] args)
 {
  Thread th;
  (th = new Thread(() -> nextPrime(10000000))).start();
  System.out.println(prime);
 }

 private static void nextPrime(long t)
 {
  if (t < 2) prime = 2;
  else for (prime = t % 2 == 0 ? t + 1 : t; !isPrime(prime); prime += 2);
 }

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

The method *nextPrime()* determines the first prime number that is greater than or equal to the parameter *t*. It is a method that I have shown before, and here it is interesting that if the parameter *t* is large, it may take a long time to perform the method. In *main()* is created a thread that performs the method, and then print the result as the value of the variable *prime*. The program will print 0, which of course is wrong. The reason is that the primary thread continues after the thread *th* is started and print the value of *prime* within the thread has updated the variable. The primary thread does not wait for the thread *th* to terminate.

The problem can be solved by changing the *main()* method to the following:

```
public static void main(String[] args)
{
 Thread th;
 (th = new Thread(() -> nextPrime(10000000))).start();
 try
 {
  th.join();
 }
 catch (Exception ex)
 {
 }
 System.out.println(prime);
}
```

A thread has a method called *join()*. If another thread – and here is the primary thread – is performing the method *join()* on a *Thread* object, it means that the second thread must wait until the first thread terminates. In this case, this means that the last print statement is not performed before the thread *th* has calculated the result and terminates.

EXERCISE 2

In this exercise, you should use the methods *nextPrime()* and *isPrime()* from the above example, the first to be modified slightly.

Create a new project, you can call *JoinThreads*. Start by adding the following class, which are nothing more than a simple encapsulation of a *long*:

```
class Prime
{
 private long value;

 public long getValue()
 {
  return value;
 }

 public void setValue(long value)
 {
  this.value = value;
 }
}
```

You must then add the following class, which defines a *Runnable* object:

```
class Creator implements Runnable
{
 private long value;
 private Prime prime;

 public Creator(long value, Prime prime)
 {
  this.value = value;
  this.prime = prime;
 }

 public void run()
 {
  nextPrime(value, prime);
 }

 public static void nextPrime(long t, Prime prime)
 {
 }
}
```

The method *nextPrime()* is, in principle, the same method as the method in the above example, but it now has an additional parameter. The method must set the value of the object to the first prime number that is greater than or equal to *t*. Note that it is still a *static* method and it is placed in this class because of the method *run()* that refer to *nextPrime()*.

Then write the following *main()* method, which determines 13 "large" primes and print them on the screen:

```
public static void main(String[] args)
{
 long[] values =
  { 1000000, 10000000, 100000000, 1000000000, 10000000000L, 100000000000L,
    1000000000000L, 10000000000000L, 100000000000000L, 1000000000000000L,
    10000000000000000L, 100000000000000000L, 1000000000000000000L };
 Prime[] primes = new Prime[values.length];
 for (int i = 0; i < primes.length; ++i) primes[i] = new Prime();
 for (int i = 0; i < values.length; ++i) Creator.nextPrime(values[i], primes[i]);
 for (int i = 0; i < primes.length; ++i) System.out.println(primes[i].getValue());
}
```

The most important is the statement, which calls the method *nextPrime()* 13 times. You must now change the *main()* method, so each call of the method *nextPrime()* is performed in its own thread. You can of course use the class *Creator* to create *Runnable* objects.

After making this changes, try if you can observe a time differences in terms where the prime numbers are generated sequentially or in separate threads.

3 SYNCHRONIZATION OF THREADS

As shown in the initial examples it is easy to create and use threads in a program, and so long that the individual threads do not share resources, the use of threads are also without major problems, but if threads share common resources, they can cause concurrency problems that must be solved by the programmer, and it does absolutely not to be simple. The typical problem is the so-called *race conditions*, where the result of a calculation depends on the timing of the threads access to a shared resource, and the classic example is that two or more threads use a common data element (a variable), and at least one of the threads change the value of the variable, but without the threads coordinates their use of the data item. As an example, consider the following method:

```
public int getId()
{
 return counter++;
}
```

where *counter* is an instance variable, and such a method is not thread safe. Apparently, the method performs only one operation, but there are several

1. save a copy of the variable *counter*
2. add 1 to the value og the variable
3. save the result in *counter*
4. returns the value, that has been saved

If, you for example imagine that two threads that calls this method, and *counter* has the value 10, the following can happen: The first thread performs the first of the above instructions, after which it is interrupted. Next comes the second thread and performs the first three operations. Then, the *counter* has the value 11. If the second thread now is interrupted, and the first thread comes back and finished its work, it will update *counter* so that it has the value 11 (and thus overwrite the changes that the other thread has done). The result is that both threads will return 11, which is an incorrect result, and the value of *counter* is incorrect. The conclusion is the need for a mechanism that can control that the threads are finished their work, or at least that another thread may not use the method before the first is completed.

To increase performance using both the Java virtual machine (JVM) and the operating system cache memories, where they keep copies of variables. This can also cause problems, since each thread possible has their own copy of a variable. If a thread modifies a variable, there is a risk that it is merely the copy that is updated, and without the other threads being aware of it, and they updates their own copies. It is also an issue that multithreaded applications must relate to.

The solution to these challenges is synchronization and locks, which ensures that the two or more threads can not simultaneously execute a critical region, which is a number of operations to be performed serially. Sometimes people talk about mutual exclusion as a thread is prevented to perform a critical region, as another thread is performing. A critical region is defined by the word *synchronized*, and in addition to ensure that other threads can not access the critical region it also ensures that variables values are synchronized with memory, when the region ends. Each Java object is associated with a so-called monitor, which has a lock which only one thread may have, and if a thread has a lock, other threads can not apply the critical region before the thread that has the lock is releasing it. There is only one thread which can have the lock, and if a different thread attempts to get the lock, it will be blocked until the first release the lock.

Consider the following program:

```
package thread05;
public class Thread05
{
 private static ID id = new ID();

 public static void main(String[] args)
 {
  for (int i = 0; i < 2; ++i)
   (new Thread(() ->
    { while (id.getValue() < 10)
     System.out.println(String.format(
      "[%d] %d", Thread.currentThread().getId(), id.getId()));
    }
  )).start();
 }
}

class ID
{
 private int id = 1;

 public int getValue()
 {
  return id;
 }

 public int getId()
 {
  int t = id;
  for (int i = 0; i < 1000000L; ++i) Math.cos(Math.sqrt(2));
  ++id;
  return t;
 }
}
```

The program illustrates some of the problems that can occur when multiple threads are using a shared resource, that here is an *ID* object. The problem is that the same *ID* is printed several times and that some values are lost. The reason is that the operation *getId()* in the class *ID* takes a long time, and thus is interrupted, so another thread can take over. The method which returns the value of variable *id*, but also counted the variable up by 1. The method also carries out a number of operations that takes time. It should illustrate that the method performs work between the variable is read and counted, and thus the method can be interrupted during that period. The *main()* method starts two threads, where each thread iterates so long that the value of the object *id* is less than 10. Each iteration reads the *ID* object (and hence counts the value of the variable up by 1) and also at the same time prints the thread's ID and the value of the object *ID*. If you executes the program, the result could be the following:

```
[10] 1
[11] 1
[10] 3
[11] 3
[11] 5
[10] 5
[11] 6
[10] 6
```

```
[11] 7
[10] 7
[11] 8
[10] 8
[11] 9
[10] 9
```

where you can see that both threads prints the same value, while other values are ignored. The reason is that the two threads are updating a shared resource *id*, but change it without being synchronized. The problem can be solved by modify the class *ID* as follows, where the method *getId()* is now defined *synchronized*:

```
class ID
{
 private int id = 1;

 public int value()
 {
  return id;
 }

 public synchronized int getId()
 {
  int t = id;
  for (int i = 0; i < 1000000L; ++i) Math.cos(Math.sqrt(2));
  ++id;
  return t;
 }
}
```

When a method is synchronized, it means that the whole operation is indivisible and is viewed as a critical region. If a thread calls the method and the method is not used by another thread, the thread may get the lock, and if another thread then attempts to perform the method, it is put in a queue and must wait until the first thread to completes the work and releases the lock. If the method that is *synchronized*, is an instance method, the lock is associated with the object on which the method is called. If the method on the other hand is a static method the lock is associated to the *java.lang.Class* object for the class where the method belongs.

Consider the following program which initializes a variable to the square root of 2, and the print result:

```
package thread07;

public class Thread07
{
 private static double root = 0;

 public static void main(String[] args)
 {
  (new Thread(() -> { root = sqrt2(); })).start();
  System.out.println(root);
 }

 private static double sqrt2()
 {
  double y = 0;
  for (int i = 0; i < 1000000000L; ++i) y = Math.sqrt(2);
  return y;
 }
}
```

The method, which determines the square root of 2 is once again a loop whose sole purpose is that the calculation must take time. The calculation is performed in its own thread, and when the primary thread prints the result, the result will be 0, which of course is wrong. The reason is that the primary thread does not wait for the thread that performs the calculation, and the problem could of course be solved by a *join()*. However, you can also solve the problem by *synchronized* and thus with a lock:

```
package thread08;
public class Thread08
{
 private static double root = 0;
 private static final Object lock = new Object();

 public static void main(String[] args)
 {
  (new Thread(() -> { synchronized(lock) { root = sqrt2(); }})).start();
  try { Thread.sleep(10); } catch (Exception ex) {}
  synchronized(lock) { System.out.println(root); };
 }
```

```
 private static double sqrt2()
 {
  double y = 0;
  for (int i = 0; i < 1000000000L; ++i) y = Math.sqrt(2);
  return y;
 }
}
```

The variable *lock* is used for a lock, and as mentioned, there is attached a lock to any object. The first statement in *main()* creates a thread, and here is the block

```
{ root = sqrt2(); }
```

synchronized. The same applies to the block with the last print statement. When the thread starts, the block of code that call *sqrt2()* is synchronized. When the primary thread then attempts to perform the block with the print statement it will be blocked because the lock associated with the object *lock* is taken and the primary threads must wait until the method, which performs the calculation is complete and releases the lock.

You should note that the object *lock* is defined final. This ensures that the object is not cached and the threads thus always refers to the memory version. You should also note that the *main()* method performs a *sleep()*. It should not be there, but is necessary, else you can risks that the primary thread takes the lock before the calculating thread takes it.

EXERCISE 3

Create a project that you can call *BufferProgram*. In the book Java 4 there is a project called *Generic*. The project has a class called *Buffer*. The class represents a generic circular buffer. Copy this class to the current project. It is necessary with a few changes:

1. The class implements an interface. Remove the implements part, such the class does not implement any interface.
2. The constructor calls a method *createArray()*. This method is not part of the class buffer. Extend the class *Buffer* with a corresponding method.

Write the following *main()* program:

```
package bufferprogram;

public class BufferProgram
{
 private static Buffer<Integer> buffer = new Buffer(10);

 public static void main(String[] args)
 {
  (new Thread(new Runnable()
  {
   public void run()
   {
    for (int n = 1; ;)
     if (!buffer.full()) try { buffer.insert(n++); } catch (Exception ex) {}
   }
  })).start();
  (new Thread(new Runnable()
  {
   public void run()
   {
    while (true)
     if (!buffer.empty())
      try { System.out.println(buffer.remove()); } catch (Exception ex) {}
   }
  })).start();
 }
}
```

That is, a program that starts two threads, where the one fills numbers in a buffer, while the other removes the numbers from the buffer. Both threads will run indefinitely. Test the program. You will find that the program starts fine, but after a short time it goes to a halt. The reason is that the class buffer is not thread safe.

You now need to modify the class *Buffer*, so it is thread safe. You can do this by defining all the *public* methods *synchronized*. Check if the problem persists.

4 DEADLOCK

Synchronization and locks means that threads are blocked and put in a waiting position, and it may lead to deadlock. It can happen if two threads must use two shared resources A and B. If one thread puts a lock on A, while the other thread puts a lock on B, and if the first thread then tries to get the lock on the B (what it can not), and the second thread attempts to get the lock on A (what it can not), there is a deadlock. As an example is below shown a program which provoke deadlock and the program "hangs":

```
package thread09;

public class Thread09
{
 private final Object lock1 = new Object();
 private final Object lock2 = new Object();

 private static int count = 0;

 public static void main(String[] args)
 {
  final Thread09 instance = new Thread09();
  (new Thread(() -> {
   while(true) { instance.work1(); instance.delay();} })).start();
  (new Thread(() -> {
   while(true) { instance.work2(); instance.delay();} })).start();
 }

 public void work1()
 {
  synchronized(lock1)
  {
   synchronized(lock2)
   {
    System.out.println(
     "[" + Thread.currentThread().getId() + "] " + (++count));
   }
  }
 }
```

```
 public void work2()
 {
  synchronized(lock2)
  {
   synchronized(lock1)
   {
    System.out.println(
     "[" + Thread.currentThread().getId() + "] " + (++count));
   }
  }
 }

 private void delay()
 {
  try
  {
   Thread.sleep(50);
  }
  catch (Exception ex)
  {
  }
 }
}
```

That threads can lead to deadlock does not mean that one should avoid to synchronize threads (above is a direct provocation), but it means that you have to be aware if a thread with a lock calls a method that indirectly also puts a lock.

5 STOP A THREAD

The *Thread* class has a method *stop()*, which is used to stop a running thread. This method is not secure and is defined deprecated, and you should avoid using the method. Instead you must yourself implement the necessary logic to stop the thread. There can be many solutions, but the following program shows an option:

```
package thread10;

import java.util.*;

public class Thread10
{
 public static void main(String[] args)
 {
  ArrayList<AThread> threads = new ArrayList();
  for (int i = 0; i < 5; ++i) threads.add(new AThread());
  for (AThread th : threads) th.start();
  try
  {
   Thread.sleep(1000);
  }
  catch (InterruptedException ie)
  {
  }
  for (AThread th : threads) th.stopThread();
 }
}

class AThread extends Thread
{
 private volatile boolean stopped = false;

 public void run()
 {
  while(!stopped)
   System.out.println(Thread.currentThread().getId() + " is running");
 }

 public void stopThread()
 {
  stopped = true;
 }
}
```

The principle is very simple, as the *Thread* class has a variable *stopped* and the thread is running as long as this variable is *false*. The class also has a method *stopThread()*, which makes the variable *true* and thus stops the thread. You should note that the variable *stopped* is defined volatile, which means that there is no reference to a copy of the variable in a cache.

6 WAIT() AND NOTIFY()

When a thread of one reason or another is blocked, it will lost the processor and is placed in a queue of waiting threads. A thread is blocked because one or other condition occurs (for example because it can not get a lock) and must then be notified again when the blocking condition is no longer there (for example another thread releases a lock). An object has two methods, so the thread that uses the object, can block itself, and so the thread can give waiting threads a message that they must continue. The methods are called *wait()* and *notify()*.

A *wait()* means that the current thread is blocked until another thread performs *notify()* or *notifyAll()*. There are also versions of *wait()*, where you enter a time, telling how long the thread maximum should waits and is thus a timeout. *notify()* reactivates (wakes) a waiting thread (a thread waiting on that object's lock). If there are multiple threads waiting on that object, you can not assume anything about the thread that comes to life. It is determined by the system. The method *notifyAll()* will in contrast awaken all threads waiting for the current object.

As a classic example of the use of these methods I will look at the so-called producer-consumer problem. The idea is that a thread (*producer*) update a resource while another thread (*consumer*) reads the resource's value. It must be such that the consumer reads all values, but only read them once. This requires that the two threads are synchronized.

```
package thread11;

public class Thread11
{
 public static void main(String[] args)
 {
  Shared shared = new Shared(); // the shared resource
  new Producer(shared).start();
  new Consumer(shared).start();
 }
}

// Represents a shared resource, as an encapsulation of a char.
class Shared
{
 private char value; // the resource
 // indicates where the resourcen must be updated
 private volatile boolean writeable = true;
```

```
// Updates the resource
public synchronized void setValue(char value)
{
 // if the resource can not be updated the thread performs a wait()
 // on the current object, which means that the thread is suspended
 // and release the lock, such that other threads can use the object
 while (!writeable)
  try
  {
   wait();
  }
  catch (InterruptedException ex)
  {
  }

 // updates the resource and change status on writeable
 // remark that if a thread reach the next statement it has the lock
 // and other threads can not perform setValue() or getValue()
 // before this thread terminates
 this.value = value;
 writeable = false;

 // performs a notify() on the current object
 // this means, that if other threads are waiting on the lock
 // one of them come to life
 notify();
}

// Read the shared resource.
synchronized char getValue()
{
 // if the resource is writeable and it has to be updated the current
 // thread performs a wait()
 while (writeable)
  try
  {
   wait();
  }
  catch (InterruptedException ex)
  {
  }
```

```
   // changes status on writeable and send a notifikation to other threads
   // that are waiting on this object, and one of them come to life
   writeable = true;
   notify();
   return value;
  }
}

// Represents the producer
class Producer extends Thread
{
 private final Shared shared; // reference to the shared resourc

 public Producer(Shared shared)
 {
  this.shared = shared;
 }

 public void run()
 {
  for (char ch = 'A'; ch <= 'Z'; ++ch) shared.setValue(ch);
 }
}
```

```
// Represents the consumer
class Consumer extends Thread
{
 private final Shared shared; // reference to the shared resourc

 public Consumer(Shared shared)
 {
  this.shared = shared;
 }

  public void run()
  {
   char ch;
   do
   {
    System.out.print(ch = shared.getValue());
   }
   while (ch != 'Z');
   System.out.println();
 }
}
```

The program prints a line consisting of the letters from A to Z – in the right order! The comments should explain how the program works, but you should notice a single thing. The methods *getValue()* and *setValue()* in the class *Shared* performs a *wait()*. This method is always performed depending on a condition and a call of the method must be done in a loop, such the condition is tested before and after the *wait()* method. Otherwise there is a risk that the *wait()* blocks a thread so that it never comes to life again. You should also note that the *wait()* and *notify()* are always used in pairs.

EXERCISE 4

You must write a program that you can call *ProducerConsumer1*. The program has to illustrate the same problem as above, but there must be two differences:

1. the producer must "produce" all characters with code from 33–127 (and not just the characters from A to Z)
2. There must be five consumer threads, and the result could be as shown below:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRS
TUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

EXERCISE 5

Write a program that you can call *ProducerConsumer2*, and the program must again illustrate the producer-consumer problem, but this time the class *Shared* must have three fields:

1. an object of the type *Buffer<Integer>* – the class from exercise 3
2. a *counter*, that is an integer that counts from 1
3. a variable *sum*, that can accumulate a sum

In addition to *get* methods for the last two variables the class *Shared* must have two methods:

1. a method *insert()*, that in the case that the buffer is not full counts the counter up by one, and inserts the value in the buffer
2. a method *add()*, that in the case where the buffer is not empty, remove a number from the buffer and adds that number to the variable *sum*

The *Producer* class, should run in a loop, where it calls the method *insert()* and places the numbers 1 to 1000000 (both incl.) in the buffer.

The *Consumer* class should run in a loop and call the method *add()* that accumulates the sum, and it must run until there is no longer a producer.

Write the program so that there are 10 consumer objects and 5 producer objects, and when the program must complete to print the accumulated sum. If the program is performed correctly, the result shall be:

```
500000500000
```

7 TIMERS

A timer is an encapsulation of a thread, and a program can start a timer that calls a method from a certain time and possibly let the timer call the method again after a certain interval. Timers are represented by two classes. The first is the class *TimerTask*, which defines the method that the timer will perform, and the other class is *Timer* that is the timer and has methods to start and terminate the timer. The following program starts a timer, and then the program prints a message on the screen every half second. The timer is ticking for 2 seconds and then it stops the program.

```
package thread12;

import java.util.*;

public class Thread12
{
 public static void main(String[] args)
 {
  TimerTask task = new TimerTask()
  {
```

```
   public void run()
   {
    System.out.println("Alarm: The machine boils...");
    System.exit(0);
   }
  };
  Timer timer = new Timer();
  timer.schedule(task, 2000);
  for (;;)
  {
   System.out.println("Hello world");
   delay(500);
  }
 }

 private static void delay(int time)
 {
  try
  {
   Thread.sleep(time);
  }
  catch (Exception ex)
  {
  }
 }
}
```

The *TimerTask* object which defines the method to be performed, is instantiated based on an anonymous class which inherits the class *TimerTask*. The class implements the method *run()*, which is the method used by the timer thread. The *Timer* object has the type *Timer*, and the object starts the timer with the statement

```
timer.schedule(task, 2000);
```

which means that the task object's *run()* method must be carried out after 2 seconds. If you performs the program, the result could be:

```
Hello world
Hello world
Hello world
Hello world
Alarm: The machine boils...
```

Below is another version of the program where the difference is that the timer is ticking for the first time by half a second and then every second:

```
package thread13;

import java.util.*;

public class Thread13
{
 public static void main(String[] args)
 {
  TimerTask task = new TimerTask()
  {
   public void run()
   {
    System.out.println("Alarm: The machine boils...");
   }
  };
  Timer timer = new Timer();
  timer.schedule(task, 500, 1000);
  for (int i = 0; i < 10; ++i)
  {
   System.out.println("Hello world");
   delay(300);
  }
  timer.cancel();
 }

 private static void delay(int time)
 {
  try
  {
   Thread.sleep(time);
  }
  catch (Exception ex)
  {
  }
 }
}
```

Note how to start the timer, and how to specify the start time and the timer interval for how often the timer should tick. A timer starts a thread which by default is a none-daemon thread. The program terminates therefore, only when the timer is stopped what happens in the last statement in *main()*. If you performs the program, the result could be:

```
Hello world
Hello world
Alarm: The machine boils...
Hello world
Hello world
Hello world
Alarm: The machine boils...
Hello world
Hello world
Hello world
Hello world
Alarm: The machine boils...
Hello world
```

A timer can also be started as a daemon thread what the program *Thread14* illustrates. The only thing to do is to create the timer as follows:

```
Timer timer = new Timer(true);
```

and if you do, the last *cancel()* statement is unnecessary.

PROBLEM 1

In this problem, you must write some classes to simulate an alarm that senses one or more thermometers and send messages to different display devices regarding the temperature to which the thermometers shows. You should solve the exercise according to the following guidelines.

1) Start a new project that you can call *Alarms*. Add the following class to represent a thermometer:

```
public class Thermometer
{
 private static final Random rand = new Random();
 private int value = rand.nextInt(10) + 15;

 public Thermometer()
 {
  ...
 }

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

The class has an *int* variable which represents the temperature, and it is initialized with a random value between 15 and 24 degrees.

You must write a constructor, so it starts a timer that ticks the first time by 1 second and then ticking every second. Each time the timer is ticking, it should add a random value between -5 and 10 to the variable *value*. That is, the timer changes the temperature – randomly up or down, but mostly up.

2) Add the following interface to the project, which should define the communication between an alarm and a display:

```
package alarmer;

public interface IDisplay
{
 public Thermometer getThermometer();
 public void show(int temp);
 public void warning(int temp);
 public void error();
}
```

3) As a next step you must to write a class that can represent a display that can show the temperature:

```
public class Display extends Thread implements IDisplay
{
 private Termometer therm;
 private String name;

 public Display(String name, Termometer therm)
 {
  ...
 }

 public void run()
 {
  ...
 }

 public Thermometer getThermometer()
 {
  return term;
 }

 public void show(int temp)
 {
  System.out.println(String.format("%d degrees in %s", temp, name));
 }

 public void warning(int temp)
 {
  System.out.println(String.format("Warning: %d degrees in %s", temp, name));
 }

 public void error()
 {
  System.out.println("Disaster in " + name + " ......................");
  ...
 }
}
```

A *Display* has a name, and shows the temperature of a particular thermometer, and both the name and the thermometer is initialized in the constructor. The class will represent a thread and therefore it inherits the class *Thread*, and the class' constructor has to start the thread. The *run()* method should in principle not do anything but just keep threads alive in a so call busy wait (that is a loop, which performs a short sleep()). The class implements the interface *IDisplay* and the *show()* method simply show the temperature, while the method *warning()* must warn that the temperature is about to be critical (over 80 degrees), and finally the method *error()* illustrate a disaster situation (when the temperature is 100 degrees). The method *error()* must also terminate the thread.

4) You must next add a class *Alarm*:

```
public class Alarm
{
 private ArrayList<IDisplay> observers = new ArrayList();
 private Thermometer[] therms;

 public Alarm(Thermometer ... therms)
 {
  ...
 }
```

```
 public void addObserver(IDisplay observer)
 {
  observers.add(observer);
 }
}
```

The class should have two variables, the first is a collection of objects of the type *IDisplay* that will receive messages regarding the thermometers state. The class must therefore have a method *addObserver()*, which can add an observer to the list. The second variable is an array with *Thermometer* objects, which are the thermometers that the alarm must monitor. The constructor shall initialize the array and start a timer that is ticking the first time after 500 milliseconds, and otherwise ticks every 800 milliseconds. Each time the timer is ticking, it should for each thermometer send messages to the observers that concerning the thermometer when to call *error()*, if the temperature is greater than or equal to 100 degrees and call *warning()* if the temperature is over 80 degrees and otherwise call *show()*.

5) Finally, you can test your classes with the following *main()* program:

```
package alarms;

public class Alarms
{
 public static void main(String[] args)
 {
  Thermometer[] terms =
   { new Thermometer(), new Thermometer(), new Thermometer() };
  Alarm alarm = new Alarm(terms);
  alarm.addObserver(new Display("The living room", terms[0]));
  alarm.addObserver(new Display("The office", terms[1]));
  alarm.addObserver(new Display("The winter garden", terms[0]));
  alarm.addObserver(new Display("TThe bedroom", terms[2]));
 }
}
```

8 CONCURRENCY TOOLS

Multithreaded applications can generally lead to concurrency issues when multiple threads need access to the same resources, and where the order of the threads access to the resources plays a role. This requires the threads to be synchronized, and this may in turn cause other problems as deadlock and stavation in which a thread will not be running, since it all the time is interrupted. As shown in the previous chapter, the basic solution is critical regions defined with synchronized code, and the methods *wait()* and *notify()*. These operations are very basic, and although in principle is simple enough, it's not so easy to use the synchronization operations and not to use them correctly. Therefore, Java has defined some classes, which aims to make it easier to write multithreaded applications, and partly to make the programs more effective. Note in this context that the use of *synchronized* on the one hand ensure critical regions, but also easily lead to programs with poor performance, where threads are too often interrupted and have to wait.

8.1 EXECUTORS

The concurrency tools include an *executor* as an alternative to starting a thread. It is an object that directly or indirectly implements the interface *Executor*, and the purpose is partly to decoupling start of a *task* from the execution, and also make it more efficient to start a thread and make more facilities available for threads. Here is a task an object that implements the interface *Runnable* or the interface *Callable*. The interface *Executor* does not define all the desired properties regarding threads, and therefore there is defined an extended interface, called *ExecutorService*. The principle is that you start a thread in the following way:

```
Executor executor = ...;
executor.execute(new RunnableTask());
```

The following program will partly show how to use an *Executor*, and partly what a *Callable* object is:

```
package thread15;

import java.math.*;
import java.util.concurrent.*;

public class Thread15
{
 public static void main(String[] args)
 {
```

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Callable<BigDecimal> callE = new ECalculator(200);
Callable<Long> callP = new PrimeCalculator(1000000000000000L);
Future<BigDecimal> eTask = executor.submit(callE);
Future<Long> pTask = executor.submit(callP);
try
{
 while (!eTask.isDone() || !pTask.isDone())
 {
  System.out.println("waiting");
  try
  {
   Thread.sleep(100);
  }
  catch (Exception ex)
  {
  }
 }
 System.out.println(eTask.get());
 System.out.println(pTask.get());
}
catch(Exception ex)
{
```

```
    System.err.println(ex);
   }
   executor.shutdownNow();
  }
}

class ECalculator implements Callable<BigDecimal>
{
 private int dec;

 public ECalculator(int dec)
 {
  this.dec = dec;
 }

 public BigDecimal call()
 {
  MathContext mc = new MathContext(dec, RoundingMode.HALF_UP);
  BigDecimal y = BigDecimal.ZERO;
  for (int i = 0; ; ++i)
  {
   BigDecimal fac = BigDecimal.ONE.divide(factorial(new BigDecimal(i)), mc);
   BigDecimal z = y.add(fac, mc);
   if (z.compareTo(y) == 0) break;
   y = z;
  }
  return y;
 }

 private BigDecimal factorial(BigDecimal n)
 {
  return n.equals(BigDecimal.ZERO) ?
   BigDecimal.ONE : n.multiply(factorial(n. subtract(BigDecimal.ONE)));
 }
}

class PrimeCalculator implements Callable<Long>
{
 private long n;

 public PrimeCalculator(long n)
 {
  this.n = n;
 }
```

```
 public Long call()
 {
  long t = n;
  if (t <= 2) return new Long(2);
  if (t % 2 == 0) ++t;
  while (!isPrime(t)) t += 2;
  return t;
 }

 private 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 = (long)Math.sqrt(n) + 1; t <= m; t += 2)
   if (n % t == 0) return false;
  return true;
 }
}
```

The program starts two tasks, performed in separate threads besacuse it is time-consuming operations, where the two tasks respectively determines a large prime number, and the number *e*. The class *PrimeCalculator* defines a task as a *Callable* object that determines the first prime number greater than or equal to a parameter passed to the constructor. Immediately a *Callable* class looks like a *Runnable* class, and both interfaces defines a method (respectively *call()* and *run()*), which is the method that is performed when a corresponding object is started as a thread, but *Callable* is a generic interface and the method *call()* returns a value whose type is the parameter type. This value is returned when the thread terminates. The class *ECalculator* are in principle identical. The interface also implements *Callable* (but parameterized with another type). The constructor has a parameter *dec*, and the *call()* method determines the number e with *dec* decimals.

Then there is the *main()* program, and the first thing that happens is the creation of a *ExecutorService* object. This is done by calling a static method in the class *Executors*, which creates an *Executor* object with a *thread pool* that can accommodate two threads. Such a thread pool efficiency the creation of threads. Then on the basis of the above are instantiated two *Callable* objects, that starts two threads, for example

```
Future<BigDecimal> eTask = executor.submit(callE);
```

Here you should note the type *Future* that is a generic interface, and in this case an *eTask* object that can be used to refer to the thread's return value. In this case, the primary thread performes a busy wait until both threads terminates, after which the results are printed.

You should note the last statement in *main()*, which close the *Executor*. It is necessary, otherwise there will be registered a none-daemon thread, and the program will then not terminate.

EXERCISE 6

Create a project that you can call *Calculations*. Copy the two classes *ECalulator* and *PrimeCalculator* to the project. You must change the two classes so

1. instead of *Callable* they implements the interface *Runnablede*
2. they do not return a value, but instead prints the results on screen

You must then add a class:

```
class SqrtCalculator implements Runnable
{
 private BigDecimal x;
 private int dec;
```

```
 public SqrtCalculator(BigDecimal x, int dec) throws IllegalArgumentException
 {
  if (x.signum() < 0) throw new IllegalArgumentException(
   "sqrt(): The argument must be none-negative");
  this.x = x;
  this.dec = dec;
 }

 public void run()
 {
 }
}
```

where the *run()* method should print the square root of x.

Finally, write the following *main()* method:

```
public static void main(String[] args)
{
 // creates an ExecutorService object with room for 5 running threads

// loop that submits 100 runnable objects to the executor object
// when there for every iteration randomly are instantiated an
// object of the classes ECalculator, PrimeCalculator and SqrtCalculator
for (int i = 0; i < 100; ++i)
{
}

 // the primary thread has to wait so long that all threads are executed

 // termnates the executor object
}
```

When you are done, you may experimenting in terms of how many threads that can simultaneously run and how many tasks the loop submits.

8.2 COUNTDOWNLATCH

A *CountDownLatch* is a synchronization object, which allows one or more threads to wait at a "gate" until another thread opens the gate, after which the waiting threads can proceed. A *CountDownLathch* is primary a counter and has operations, where a thread waits until the counter is counting down to the 0.

The following program will show how to apply a *CountDownLatch* to synchronization of objects. The program starts N worker threads, that are unable to continue until a *CountDownLatch* is countdown. After the primary thread has done a work, it is blocked until each worker thread has counted down the *CountDownLatch*. This means that the primary thread waits for all worker threads to terminate.

```
package thread16;

import java.util.*;
import java.util.concurrent.*;

public class Thread16
{
 private final static int N = 5;

 public static void main(String[] args)
 {
  final CountDownLatch startLatch = new CountDownLatch(1);
  final CountDownLatch doneLatch = new CountDownLatch(N);
  Runnable worker = new Worker(startLatch, doneLatch);
  ExecutorService executor = Executors.newFixedThreadPool(N);
  for (int i = 0; i < N; ++i) executor.execute(worker);
  try
  {
   System.out.println("Main working");
   Thread.sleep(1000);
   startLatch.countDown();
   System.out.println("Main doing work");
   doneLatch.await();
   executor.shutdownNow();
  }
  catch (InterruptedException ex)
  {
   System.err.println(ex);
  }
 }
}

class Worker implements Runnable
{
 private static final Random rand = new Random();
 private final CountDownLatch start;
 private final CountDownLatch done;
```

```
public Worker(CountDownLatch start, CountDownLatch done)
{
 this.start = start;
 this.done = done;
}

public void run()
{
 try
 {
  print("Entered run()");
  start.await();
  print("Working");
  Thread.sleep(rand.nextInt(1000));
  done.countDown();
 }
 catch (InterruptedException ex)
 {
  System.err.println(ex);
 }
}
```

```
 void print(String text)
 {
  System.out.println("[" + Thread.currentThread().getId() + "] " + text);
 }
}
```

The class *Worker* defines a runnable object (a task) to simulate that work is carried out on the machine. An object is initialized with two *CountDownLatch* objects. The *run()* method starts to print a message that the threads are started, but then block itself (at the gate):

```
start.await();
```

and waiting for another thread to signal that it must continue. When that happens, the thread prints again a text and performs a *sleep()* (to illustrate that the thread do something), and when it is finished the second synchronization object is counting down so it knows when all threads are finished.

main() defines two synchronization objects:

```
final CountDownLatch startLatch = new CountDownLatch(1);
final CountDownLatch doneLatch = new CountDownLatch(N);
```

where the first should be used to block the start of the N worker threads until the primary thread opens up and allows the threads to continue. The second should be used to get the primary thread to wait until all worker threads are completed. Next is created a runnable object to a worker thread, and there is created an *Executor* object for N threads. The next loop adds N threads to the executor object. Note that the threads are started on basis of the same object. The threads start immediately, but when they blocks themself, and there is no momentum prior to the primary thread opens up. It first performs a work and then opens the gate (the synchronization object *startLatch* is counted down), and the worker threads can proceed. The primary thread also continues and performs

```
doneLatch.await();
```

This means that the primary thread is blocked waiting until the worker threads have counted the synchronization object down to 0.

EXERCISE 7

Start by creating a copy of the project *Calculations* (Exercise 6), and calls for example the copy for *Calculations1*. Open the copy in NetBeans. The solution has a problem as the primary thread waits to all the worker threads are finish by waiting long enough. Solve this problem by using a *CountDownLatch*.

8.3 CYCLICBARRIER

A *CyclicBarrier* is a synchronization object which can be used to ensure that multiple threads are waiting on each other at a certain barrier point. The object is cyclic, since it can be recycled after having released waiting threads. The synchronization mechanism is useful if a program that starts a certain number of threads which must wait for each other. Sometimes you can think of it as multiple threads are performing work on a common problem, and that the task is first solved when all the threads have completed their work.

The following program creates and initalizes a square matrix. Next, the application determine the square root of each of the matrix's items, but it must take place in that way, that starting a thread for each row, wherein the thread is processing that row. The result is only available when all threads have finished their work, after which the individual threads work in principle are merges into a finished result. It obviously can not be made until the individual threads are completed and it can be synchronized with a *CyclicBarrier*.

```
package thread17;

import java.util.concurrent.*;

public class Thread17
{
 private static final int N = 5; // the size of the matrix

 public static void main(String[] args)
 {
  float[][] matrix = create();
  print(matrix);
  System.out.println();
  Solver solver = new Solver(matrix);
  System.out.println();
  print(matrix);
 }
```

```
 private static float[][] create()
 {
  float[][] matrix = new float[N][N];
  for (int r = 0, n = 0; r < matrix.length; ++r)
   for (int c = 0; c < matrix[r].length; ++c) matrix[r][c] = ++n;
  return matrix;
 }

 private static void print(float[][] matrix)
 {
  for (int r = 0; r < matrix.length; ++r)
  {
   for (int c = 0; c < matrix[r].length; ++c) System.
   out.printf("%10.4f", matrix[r][c]);
   System.out.println();
  }
 }
}

class Solver
{
 private final float[][] data; // matricen
 private final CyclicBarrier barrier; // synkroniserer flere trådes arbejde
```

```
public Solver(float[][] matrix)
{
 data = matrix;
 barrier =
  new CyclicBarrier(matrix.length, new Runnable()
  { public void run() { merge(); } });
 for (int r = 0; r < matrix.length; ++r) new Thread(new Worker(r)).start();
 synchronized("abc")
 {
  try
  {
   System.out.println("[" +Thread.currentThread().getId() + "] waiting");
   "abc".wait();
   System.out.println("[" +Thread.currentThread().getId() + "] notified");
  }
  catch (InterruptedException ie)
  {
   System.out.println("main thread interrupted");
  }
 }
}

void merge()
{
 System.out.println("merging");
 synchronized("abc")
 {
  "abc".notify();
 }
}

class Worker implements Runnable
{
 private final int row; // the row that the thread works on
 private boolean done = false; // when the thread has to stop

 public Worker(int row)
 {
  this.row = row;
 }

 public boolean done()
 {
  return done;
 }
```

```
  private void work()
  {
   System.out.println("Behandler række: " + row);
   for (int i = 0; i < data[row].length; ++i)
    data[row][i] = (float)Math.sqrt(data[row][i]);
   done = true;
  }

  public void run()
  {
   while (!done())
   {
    work();
    try
    {
     barrier.await();
    }
    catch (Exception ex)
    {
     return;
    }
   }
  }
}
```

The method *create()* creates the matrix and the method *print()* prints the matrix on the screen. Both of these methods are trivial. After the matrix has been created, is instantiated an object of the type *Solver*, and it is the object which solves the problem. The class *Solver* creates a *CyclicBarrier* which is initialized with a number of worker threads, and a runnable object to be performed when all worker threads have completed their work. In this case, the object's *run()* method calls the method *merge()*. After the worker threads are created – one thread for each row – and the threads are created on basis of a runnable object of the type *Worker*. The class is basically trivial, but it is an inner class (in the class *Solver*) and therefore it knows the synchronization object *barrier*, and after it has done its job, it performs a

```
barrier.await();
```

which means that the thread is waiting at the barrier. After the constructor in the class *Solver* has started the worker threads, it performs *wait()* on a lock (which is simply a string), and it means that the primary thread waits until it gets a notify from another thread.

After all worker threads are performed an *await()* on the barrier (all threads have reached the barrier), the barrier object let the threads continue (which here simply means that they stop) and carry out its own thread, which here calls the *merge()* method. It does not much, but it sends a *notify()* on the lock set by the primary thread, and the result is that the primary thread continues.

If you executes the program, you get the result:

```
  1,0000    2,0000     3,0000     4,0000     5,0000
  6,0000    7,0000     8,0000     9,0000    10,0000
 11,0000   12,0000    13,0000    14,0000    15,0000
 16,0000   17,0000    18,0000    19,0000    20,0000
 21,0000   22,0000    23,0000    24,0000    25,0000

Behandler række: 0
Behandler række: 1
Behandler række: 2
Behandler række: 3
[1] waiting
Behandler række: 4
merging
[1] notified
```

```
1,0000      1,4142      1,7321      2,0000      2,2361
2,4495      2,6458      2,8284      3,0000      3,1623
3,3166      3,4641      3,6056      3,7417      3,8730
4,0000      4,1231      4,2426      4,3589      4,4721
4,5826      4,6904      4,7958      4,8990      5,0000
```

PROBLEM 2

Create a new project, that you can call *Calculations2*. Add the following class which has a static method that creates a random *BigDecimal* with 100 digits before and after the decimal point, as well as a static method, which determines the square root of a *BigDecimal*:

```
abstract class Calc
{
 private static final int N = 100;
 public static final Random rand = new Random();
 public static final MathContext context =
  new MathContext(N, RoundingMode.HALF_UP);

 public static BigDecimal random()
 {
  StringBuilder builder = new StringBuilder();
  builder.append((char)('1' + rand.nextInt(9)));
  for (int i = 0; i < N; ++i) builder.append((char)('0' + rand.nextInt(10)));
  builder.append('.');
  for (int i = 0; i < N; ++i) builder.append((char)('0' + rand.nextInt(10)));
  return new BigDecimal(builder.toString());
 }

 public static BigDecimal sqrt(BigDecimal x)
 {
  BigDecimal value = BigDecimal.ZERO;
  if (x.signum() != 0)
  {
   BigInteger t = x.movePointRight(N << 1).toBigInteger();
   BigInteger y = t.shiftRight((t.bitLength() + 1) >> 1);
   for (BigInteger z = y;; z = y)
   {
    y = y.add(t.divide(y)).shiftRight(1);
    if (y.compareTo(z) == 0) break;
   }
   value = new BigDecimal(y, N);
  }
  return value;
 }
```

You do not have to study the code when the only goal is that the method *sqrt()* that takes a long time.

Add a method to the main class, which creates an array of *n* objects of type *BigDecimal* when the objects must be created with the method *Calc.random()*:

```
private static BigDecimal[] create(int n)
{
}
```

The program should now use this method to create an array of the type *BigDecimal* with 10 numbers, and then the program must print the result of method

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

when the method must return the sum of the square roots of the numbers in the array. The method must also print the time of the calculation measured in nanoseconds. You can read the hardware clock with:

```
System.nanoTime()
```

When the method *sum1()* works, write a similar method *sum2()*, but it must instead works that way, that it starts a worker thread to calculate each square root. It can only calculate the sum, and after all threads have calculated the square roots, you need to synchronize the threads using a *CyclicBarrier*.

Examine which of the two methods that have the best performance. Also explore what happens if you change the number of digits in the class *Calc* (the constant *N*). Also check what happens if you change the number of threads and thus the size of the array.

8.4 EXCHANGER

An *Exchanger* provides a synchronization point at which threads can exchange objects. That is, that a thread can call an *exchange()* methode and wait for another thread to do the same, and then the threads can switch two objects. Maybe it's not the most commonly used synchronization object, but it has its uses in connection with the implementation of specific algorithms. The following program shows how to use an *Exchanger*, where two threads swap two objects of the type *Buffer*. The class *Buffer* is the same class that is used in some of the above exercises.

```
package thread18;

import java.util.*;
import java.util.concurrent.*;

public class Thread18
{
 private static final Random rand = new Random();
 private final static Exchanger<Buffer<Character>> exchanger = new Exchanger();
 private final static Buffer<Character> buffer1 = new Buffer(15);
 private final static Buffer<Character> buffer2 = new Buffer(10);
 private static char ch = 'Z';

 public static void main(String[] args)
 {
  class Producer implements Runnable
  {
   public void run()
   {
    Buffer<Character> buffer = buffer1;
    try
    {
     while (true)
     {
```

```
       buffer.insert(next());
       Thread.sleep(rand.nextInt(500));
       if (buffer.full()) buffer = exchanger.exchange(buffer);
      }
     }
     catch (Exception ex)
     {
      System.out.println(ex);
     }
    }
   }

   class Consumer implements Runnable
   {
    public void run()
    {
     Buffer<Character> buffer = buffer2;
     try
     {
      while (true)
      {
       Thread.sleep(rand.nextInt(500));
       if (buffer.empty())
       {
        buffer = exchanger.exchange(buffer);
        System.out.println();
       }
       System.out.print(buffer.remove());
      }
     }
     catch (Exception ex)
     {
      System.out.println(ex);
     }
    }
   }
   new Thread(new Producer()).start();
   new Thread(new Consumer()).start();
  }

  private static char next()
  {
   if (ch == 'Z') ch = 'A'; else ++ch;
   return ch;
  }
}
```

8.5 SEMAPHORE

A *semaphore* manages a number of permits in relation to a number of threads that want to access a shared resource. A thread that is trying to obtain permission for the resource, while all licenses are used, is blocked until another thread releases the license. A *semaphore's* value is an integer whose value is not negative and can be counted up or down. If the semaphore can take only the values 0 and 1, we speak of a *binary semaphore*, and may the value be greater than 1, it is called a *counting semaphore*. This means that is the access to a resource controlled with a counting semaphore, there are several threads at a time (determined by semaphoren) that can access the resource. The following program will show how to use a semaphore:

```
package thread19;

import java.util.*;
import java.util.concurrent.*;

public class Thread19
{
 public static void main(String[] args)
 {
  final Resources resources = new Resources();
  Worker worker = new Worker(resources);
  int N = 2 * Resources.N;
  ExecutorService executor = Executors.newFixedThreadPool(N);
  for (int i = 0; i < N; ++i) executor.execute(worker);
  try
  {
   executor.awaitTermination(20, TimeUnit.SECONDS);
  }
  catch (Exception ex)
  {
  }
  executor.shutdownNow();
 }
}

class Worker implements Runnable
{
 private static final Random rand = new Random();
 private Resources resources;
 private long counter = 0;
```

```
public Worker(Resources resources)
{
 this.resources = resources;
}

public void run()
{
 try
 {
  while (counter < 100)
  {
   Resource res = resources.getResouce();
   System.out.printf("[%d] anvender %s: %3d\n",
    Thread.currentThread().getId(), res.toString(), res.getValue());
   Thread.sleep(500 + rand.nextInt(500));
   res.updateValue(++counter);
   resources.putResource(res);
   System.out.printf("[%d] opdateret %s: %3d\n",
    Thread.currentThread().getId(), res.toString(), res.getValue());
   Thread.sleep(500 + rand.nextInt(500));
  }
 }
 catch (InterruptedException ex)
 {
```

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

class Resources
{
 public static final int N = 10;    // number of Resource objects
 private final Resource[] resources = new Resource[N];
 private final Semaphore sema = new Semaphore(N, true);
 private final boolean[] used = new boolean[N];

 public Resources()
 {
  for (int i = 0; i < resources.length; ++i) resources[i] = new Resource();
 }

 public Resource getResouce() throws InterruptedException
 {
  sema.acquire();
  return get();
 }

 public void putResource(Resource res)
 {
  if (unused(res)) sema.release();
 }

 private synchronized Resource get()
 {
  for (int i = 0; i < N; ++i)
  {
   if (!used[i])
   {
    used[i] = true;
    return resources[i];
   }
  }
  return null;
 }

 private synchronized boolean unused(Resource res)
 {
  for (int i = 0; i < N; ++i)
  {
   if (res.equals(resources[i]))
   {
```

```
     if (used[i])
     {
      used[i] = false;
      return true;
     }
     else return false;
    }
   }
   return false;
 }
}

class Resource
{
 private static int ID = 0;
 private int id; // the ressource's id
 private long value; // the ressource's value

public Resource()
 {
  id = ++ID;
 }

 public long getValue()
 {
  return value;
 }

 public void updateValue(long value)
 {
  this.value += value;
 }

 public boolean equals(Object obj)
 {
  if (obj == null) return false;
  if (getClass() == obj.getClass()) return ((Resource)obj).id == id;
  return false;
 }

 public String toString()
 {
  return String.format("Resource: [%d]", id);
 }
}
```

The class *Resource* is a simple encapsulation of a long and has basically two methods:

1. *getValue()* that returns the object's value
2. *updateValue()*, that adds a value to the object's value

Each object is assigned also a consecutively number when it is created, and it is only to have an identification of the object and to give the object a name in *toString()*.

The most importannt class is the class *Resources*, which is an encapsulation of an array of *Resource* objects. The idea is that a thread may wish to get a *Resource* object and update it. While this happens, the object is locked and is released again after it is updated. The class creates a *Semaphore* to check the threads access to an object:

```
private final Semaphore sema = new Semaphore(N, true);
```

It is a counting semaphore, and the first parameter indicates the number of threads which can simultaneously use an object of the type *Resources*. This corresponds to the array length, since it is the number of *Resource* objects that are available. The last parameter ensures that threads are blocked on this semaphore, not being starved and never get running again. When creating a *Resources* object is the semaphore's value *N*, and thus there are *N* threads that can use the object. The class also has an array of type *boolean*, and it is used to control which *Resource* objects that are in use.

The class has two public methods: *getResource()* and *putResource()*. The first returns an available *Resource* object, and it starts to count down with semaphoren with 1:

```
sema.acquire();
```

If the semaphore is 0, the current thread blocked and must wait until another thread counts the semaphoren up. Otherwise the semaphore is counted down by 1 and the method continues. In this case, it means that the method *getResource()* call a synchronized method *get()*, there must find and returns a *Resource* object. The method *get()* must at the same time set the actual object as used. The method *putResource()* has as a parameter a *Resource* object, and the method has to release this object. To this the method calls a *synchronized* method *unused()*, and if the object is used, it is marked as *unused*, and the method returns true. If so, then the method *putResource()* performs

```
sema.release();
```

which means that the semaphore is counted up by 1, and at the same time it indicates that a blocked thread can now use the *Resources* object.

The class *Worker* is a class which defines a thread that performs work by asking for a *Resource* object. There is not much to explain, but you must note that it is essential that the *run()* method executes *putResouce()* since the semaphore otherwise not is counted up.

Back is the *main()* method, which starts a number of worker threads. Here you must notice that it starts more threads than can simultaneously use the *Resources* object (twice as many). It is to illustrate that some of the threads is blocked by the semaphore, and in turn becomes unblocked again.

EXERCISE 8

Create a project that you can call *SemaphoreProgram*. The program is an example of using both a *binary semaphore* and a *counting semaphore*.

Start by typing the following class, which is an encapsulating of a *Map<Integer, String>*, when it is a requirement that the class must be written as a singleton:

```
class Storage
{
 private static int ID = 0; // the consecutive numbering of objects
 private Map<Integer, String> cache = new HashMap();

 public int insert(String name)
 {
  // add a new name to the data structure identified by the next id
 }

 public String getName(int id)
 {
  // returns a name with an id or null, the id is not found
 }

 public int length()
 {
  // Returns the number of names
 }
}
```

The class should illustrate a data structure consisting of names, identified by a sequential number.

The goal of the exercise is to create threads that reads names (performing *getName()*) as well as threads that adds names (performing *insert()*). It should be such that up to 5 threads simultaneously read, while only a single thread can add names. It must be controlled by two semaphores. To make things a little easier, you should encapsulate the two semaphores in the below class when it is a requirement that the class must be written as a singleton:

```
class StorageLock
{
 private final Semaphore writeLock = new Semaphore(1);
 private final Semaphore readLock = new Semaphore(5);

 public void getWriteLock() throws InterruptedException
 {
  writeLock.acquire();
 }
```

```
 public void releaseWriteLock()
 {
  writeLock.release();
 }

 public void getReadLock() throws InterruptedException
 {
  readLock.acquire();
 }

 public void releaseReadLock()
 {
  readLock.release();
 }
}
```

As a next step you must write a *Writer* class that defines a *Runnable* type that can add names to the data structure *Storage*:

```
class Writer implements Runnable
{
 private String[] names;
```

```
 public Writer(String[] names)
 {
  this.names = names;
 }

 public void run()
 {
  for (String name : names)
  {
   // set a write lock
   // add name
   // release the lock
   // sleep a random time less than 2 seconds
  }
 }
}
```

You must corresponding write a class *Reader* that defines a *Runnable* type that can read the names in the data structure *Storage*:

```
class Reader implements Runnable
{
 public void run()
 {
  while (true)
  {
   // set a read lock
   // read a name with a random id
   // release the lock
   // prints the name, if it is not null
   // sleep a random time less than a ½ second
  }
 }
}
```

Then there is the only main program:

```
public class SemaphoreProgram
{
 private static final String[] boys = ... // array with 10 boy names
 private static final String[] girls = ... // array with 10 girl names
```

```
 public static void main(String[] args)
 {
  // creates a thread to a Writer object which adds boy names
  // creates a thread to a Writer object which adds girl names
  // Creates 10 Reader threads, when it should be daemon threads
  // start all the threads
  // the primary thread should join the two writer threds
 }
}
```

Test the program and check that everything works as intended.

8.6 PHASER

A *Phaser* looks like a *CyclicBarrier*, but is more flexible. In the same way as with a *CyclicBarrier* you can be achieve that a number of threads are waiting at the barrier, until the last thread is arriving, but in contrast to a *CyclicBarrier*, that waits for a fixed number of threads, a *Phaser* act as a barrier for a variable number threads, and there may be several phases – barriers. The following program starts 5 threads in addition to the primary thread, and the execution of the 6 threads are synchronized by a *Phaser* object with 4 phases:

```
package thread20;

import java.util.*;
import java.util.concurrent.*;

public class Thread20
{
 public static void main(String[] args)
 {
  Phaser phaser = new Phaser(1);
  Thread thread1 = new Thread(new Worker(phaser), "Thread-1");
  Thread thread2 = new Thread(new Worker(phaser), "Thread-2");
  Thread thread3 = new Thread(new Worker(phaser), "Thread-3");
  Thread thread4 = new Thread(new Worker(phaser), "Thread-4");
  Thread thread5 = new Thread(new Worker(phaser), "Thread-5");
  System.out.println("\n-------- Start Phaser ---------------");
  thread1.start();
  thread2.start();
  thread3.start();
  thread4.start();
  thread5.start();
  work(phaser);
  work(phaser);
  work(phaser);
  work(phaser);
```

```
   phaser.arriveAndDeregister();
   if(phaser.isTerminated()) System.out.println("\nThe Phaser object terminated");
  }

  private static void work(Phaser phaser)
  {
   int phase = phaser.getPhase();
   phaser.arriveAndAwaitAdvance();
   System.out.println("------ Phase - " + phase + " is terminated ----------");
  }
}

class Worker implements Runnable
{
 private static final Random rand = new Random();
 private Phaser phaser;
 private double value = 0;
```

```
public Worker(Phaser phaser)
{
 this.phaser = phaser;
 this.phaser.register();
 System.out.println("New thred registered: " +
  Thread.currentThread().getName());
}

public void run()
{
 todo(2);
 todo(3);
 todo(5);
 result();
 phaser.arriveAndDeregister();
}

private void todo(int t)
{
 System.out.println(Thread.currentThread().getName() +
  " - has reached the barrier and works in phase " +
  phaser.getPhase() + ", Value = " + value);
 phaser.arriveAndAwaitAdvance();
 work(rand.nextInt(Integer.MAX_VALUE), t);
}

private void result()
{
 System.out.println(Thread.currentThread().getName() +
  " - has reached the barrier and works in phase " +
  phaser.getPhase() + ", Value = " + value);
 phaser.arriveAndAwaitAdvance();
}

private void delay(int time)
{
 try
 {
  Thread.sleep(time);
 }
 catch (InterruptedException e)
 {
 }
}
```

```
 private void work(long n, int t)
 {
  for (int i = 0; i < n; ++i) value = Math.sqrt(t);
 }
}
```

The class *Worker* defines the secondary threads. The constructor has a *Phaser* object as a parameter, and the constructor sign up to the Phaser object as one of the threads which the object has to wait for:

```
this.phaser.register();
```

The *run()* method executes four methods (calls the method *todo()* three times and the method *result()* once). Each of these methods carry out some work in a phase. That is that the first *todo()* is first performed when all threads have reached the first barrier, the next *todo()* when all threads have reached the next barrier and so on. If you look at each phase's methods, it is not so important, what they do, but they perform the method

```
phaser.arriveAndAwaitAdvance();
```

which means that the thread must wait at the barrier until all threads have reached. The *Phaser* object is created in the *main()* method:

```
Phaser phaser = new Phaser(1);
```

where there is registered a single thread, which is the primary thread. The primary thread creates then 5 other threads and start them and then perform the method *work()*. The important thing here is the statement

```
phaser.arriveAndAwaitAdvance();
```

which means that all threads incl. the primary thread will have to wait at a barrier until all the other threads arrives. If the program is performed the result is:

```
New thred registered: main
New thred registered: main
New thred registered: main
New thred registered: main
New thred registered: main
```

```
-------- Start Phaser ---------------
Thread-2 - has reached the barrier and works in phase 0, Value = 0.0
Thread-1 - has reached the barrier and works in phase 0, Value = 0.0
Thread-4 - has reached the barrier and works in phase 0, Value = 0.0
Thread-3 - has reached the barrier and works in phase 0, Value = 0.0
Thread-5 - has reached the barrier and works in phase 0, Value = 0.0
------ Phase - 0 is terminated ----------
Thread-2 - has reached the barrier and works in phase 1, Value = 1.4142135623730951
Thread-5 - has reached the barrier and works in phase 1, Value = 1.4142135623730951
Thread-4 - has reached the barrier and works in phase 1, Value = 1.4142135623730951
Thread-3 - has reached the barrier and works in phase 1, Value = 1.4142135623730951
Thread-1 - has reached the barrier and works in phase 1, Value = 1.4142135623730951
------ Phase - 1 is terminated ----------
Thread-4 - has reached the barrier and works in phase 2, Value = 1.7320508075688772
Thread-1 - has reached the barrier and works in phase 2, Value = 1.7320508075688772
Thread-5 - has reached the barrier and works in phase 2, Value = 1.7320508075688772
Thread-3 - has reached the barrier and works in phase 2, Value = 1.7320508075688772
Thread-2 - has reached the barrier and works in phase 2, Value = 1.7320508075688772
------ Phase - 2 is terminated ----------
Thread-2 - has reached the barrier and works in phase 3, Value = 2.23606797749979
Thread-1 - has reached the barrier and works in phase 3, Value = 2.23606797749979
Thread-3 - has reached the barrier and works in phase 3, Value = 2.23606797749979
Thread-5 - has reached the barrier and works in phase 3, Value = 2.23606797749979
Thread-4 - has reached the barrier and works in phase 3, Value = 2.23606797749979
```

8.7 LOCKS

As seen in the previous chapter, Java allows to synchronize threads so that threads can safely updating shared objects, and so that a thread's updating of an object is visible to all other threads. This is done by placing code in a critical region where you can ensure mutual exclusion by using locks. Each object is associated with a lock in the form of a monitor, which ensures that only one thread at a time, can performs the statements in a critical region. A thread can only enter a critical region and obtain the lock if the lock is not used by another thread, and if it is the current thread is blocked until another thread exits the critical region and releases the lock. This mechanism will also ensure that when a thread locks a critical region, that values of shared variables stored in memory also if necessary updates copies in a cache. The opposite happens when a thread exits the critical region and releases the lock, the cached values are written back to memory.

To facilitate the work synchronization tools defines more interfaces and classes that will make it easier to work with locks, and as an example uses the following program a lock:

```
package thread21;

import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class Thread21
{
 public static void main(String[] args)
 {
  ExecutorService executor = Executors.newFixedThreadPool(3);
  final ReentrantLock lock = new ReentrantLock();
  executor.execute(new Worker("A", lock));
  executor.execute(new Worker("B", lock));
  executor.execute(new Worker("C", lock));
  try
  {
   executor.awaitTermination(10, TimeUnit.SECONDS);
  }
  catch (Exception ex)
  {
  }
  executor.shutdownNow();
 }
}
```

```
class Worker implements Runnable
{
 private final Lock lock;
 private final String name;

 Worker(String name, Lock lock)
 {
  this.name = name;
  this.lock = lock;
 }

 public void run()
 {
  System.out.println(name + " startet");
  lock.lock();
  try
  {
   System.out.printf("%s Critical region.\n",name);
   System.out.printf("%s: Works.\n", name);
   Thread.sleep(2000);
   System.out.printf("%s Terminates.\n", name);
  }
  catch (Exception ex)
  {
  }
  finally
  {
   lock.unlock();
  }
 }
}
```

A lock is represented by the class *ReentrantLock* that implements the interface *Lock*, and the program creates such a lock:

```
ReentrantLock lock = new ReentrantLock();
```

Next the program start three threads, and each thread carry out work in a critical region that is defined with the lock. There is not much to explain, and it is in principle simple to work with a lock, but it is important to ensure that the lock is always released again, and a good place is to lock up is in a *finally* block. If the above program is performed, the result could be the following:

```
A startet
C startet
B startet
A Critical region.
A: Works.
A Terminates.
C Critical region.
C: Works.
C Terminates.
B Critical region.
B: Works.
B Terminates.
```

Basic is synchronization based on critical regions on the use of *synchronized* and *wait()* and *notify()*, and you can say that a *Lock* replaces the use of *synchronized*. Similarly, an object of the type *Condition* is an object that replaces the use of *wait()* and *notify()*. Below is shown a program that solves the producer-consumer problem in which there are two threads, respectively updating and reading a shared resource, but this time the problem is solved by a *ReentrantLock* and a *Condition* object:

```
package thread22;

import java.util.concurrent.locks.*;

public class Thread22
{
 public static void main(String[] args)
 {
  Shared shared = new Shared();
  new Producer(shared).start();
  new Consumer(shared).start();
 }
}

class Shared
{
 private char value; // the shared resource
 private volatile boolean ok = false;
 private final Lock lock = new ReentrantLock();
 private final Condition condition = lock.newCondition();

 public Lock getLock()
 {
  return lock;
 }

 public char getValue() throws InterruptedException
 {
  lock.lock();
  try
  {
   while (!ok) condition.await();
   ok = false;
   return value;
  }
  finally
  {
   condition.signal();
   lock.unlock();
  }
 }

 public void setValue(char value) throws InterruptedException
 {
  lock.lock();
  try
  {
```

```
    while (ok) condition.await();
    this.value = value;
    ok = true;
   }
   finally
   {
    condition.signal();
    lock.unlock();
   }
  }
}

class Producer extends Thread
{
 private final Shared shared;

 Producer(Shared shared)
 {
  this.shared = shared;
 }

 public void run()
 {
  for (char ch = 'A'; ch <= 'Z'; ++ch)
  {
   try
   {
    shared.setValue(ch);
   }
   catch (InterruptedException ex)
   {
   }
  }
 }
}

class Consumer extends Thread
{
 private final Lock lock;
 private final Shared shared;

 Consumer(Shared shared)
 {
  this.shared = shared;
  lock = shared.getLock();
 }
```

```
 public void run()
 {
  char ch = ' ';
  do
  {
   try
   {
    lock.lock();
    System.out.print(ch = shared.getValue());
   }
   catch (InterruptedException ex)
   {
   }
   finally
   {
    lock.unlock();
   }
  }
  while (ch != 'Z');
  System.out.println();
 }
}
```

The program is similar to the previous solution, but you should notice how the class *Shared* creates a *Condition* object using the *Lock* object. This object is used in both *getValue()* and *setValue()* to perform *await()*, and within the critical region the method performs a *signal()* corresponding to the *notify()*.

EXERCISE 9

In this exercise you have to solve the same task as in exercise 5, but this time instead of using *synchronized*, *wait()* and *notify()* you must use the types *Lock* and *Condition*.

8.8 READWRITELOCK

Many times, a data structure is read more often than it needs to be adjusted, and you can have multiple threads that simultaneously read the data structure, while only one or a few threads need to update it. Accordingly, there is a lock, called a *ReadWriteLock* that ensures that multiple threads can simultaneously read a data structure, while only a single can update. The following program creates a thread and initializes a *Map<String, String>* with the Danish zip codes where the code is the key. In addition, the program creates three methods that read the data structure and print an item on the screen. To synchronize the threads access to the data structure is used a *ReadWriteLock*:

```
package thread23;

import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class Thread23
{
 private static final ReadWriteLock lock = new ReentrantReadWriteLock(true);
 private static final Map<String, String> dictionary = new HashMap();
 private static final List<String> keys = new ArrayList();
 private static final Random rand = new Random();

 public static void main(String[] args)
 {
  new Thread23().doWork();
 }

 private void doWork()
 {
  ExecutorService executor = Executors.newFixedThreadPool(4);
  executor.submit(new Writer());
  executor.submit(new Reader());
```

```
 executor.submit(new Reader());
 executor.submit(new Reader());
}

class Reader implements Runnable
{
 public void run()
 {
  while (true)
  {
   lock.readLock().lock();
   try
   {
    System.out.println("[" + Thread.currentThread().getId() + "] " +
     dictionary.get(keys.get(rand.nextInt(keys.size()))));
   }
   finally
   {
    lock.readLock().unlock();
   }
   try
   {
    Thread.sleep(100);
   }
   catch (InterruptedException ie)
   {
   }
  }
 }
}

class Writer implements Runnable
{
 public void run()
 {
  for (int i = 0; i < data.length; ++i)
  {
   lock.writeLock().lock();
   try
   {
    String[] elems = data[i].split(";");
    dictionary.put(elems[0], elems[0] + " " + elems[1]);
    keys.add(elems[0]);
   }
   finally
   {
    lock.writeLock().unlock();
   }
```

```
      try
      {
       Thread.sleep(200);
      }
      catch (InterruptedException ie)
      {
      }
     }
     System.out.println("Dictionary initialiseret.............................");
    }
   }

   private static String[] data =
   {
    "0800;Høje Taastrup",
    "0900;København C",
    "0999;København C",
    "1000;København K",
    "1050;København K",
    ...
    "9990;Skagen"
   };
  }
```

The zip codes are laid out in an array at the end of the program. At the start of the main class is defined a *ReadWriteLock*:

```
ReadWriteLock lock = new ReentrantReadWriteLock(true);
```

and also a *HashMap* called *dictionary*, and an *ArrayList*, that should be used for keys. The class *Writer* defines the thread to initialize the dictionary and you should notice, that it defines a critical region using the write lock. This ensures that only one thread can use the code that updates the data structure. The class *Reader* similarly defines a thread to read the data structure, which reads a data item with a random key. Here you should note that the critical region is defined by means of the read lock and thus more readers can use the critical region.

EXERCISE 10

In this exercise you have to solve the same problem as in exercise 8, but locking must be done with a *ReadWriteLock* instead of the class *StorageLock*. Start by creating a copy of the program *SemaphoreProgram* and call the new project for *ReadWriteProgram*. Create a *ReadWriteLock*:

```
ReadWriteLock lock = new ReentrantReadWriteLock(true);
```

and delete the class *StorageLock*. Adjust then the code to instead use the new lock.

8.9 COLLECTIONS

Java was from the start equipped with collection classes for the most commonly used data structures, and they were all thread safe, so they could be used safely in multithreaded applications. At the time, it was really thought, for the goal of Java was to develop an object-oriented programming language that was easy to use, and which resulted in robust programs. However, the language quickly became so popular that it was used also for the development of large-scale distributed solutions, and then performance was an issue, and it was a problem that the Java's collection classes were thread safe, especially because in most cases you do not have the need. The result was, that some new collection classes was developed (those that are discussed in this books), and they are not thread safe. Occasionally you has, however, the needs that collections are thread safe, and to solve these problems is defined some wrapper classes, which ensures that the classes can be used with multithreaded applications. These wrapper classes are simple to use and the following example shows a program, that use a thread safe *ArrayList*:

```
package thread24;

import java.util.*;
public class Thread24
{
 public static void main(String[] args)
 {
  List<Integer> list = java.util.Collections.synchronizedList(new ArrayList());
  for (int i = 1; i <= 100; ++i) list.add(i);
  int sum = 0;
  for (Integer t : list) sum += t;
  System.out.println(sum);
 }
}
```

and there are similar wrapper classes to the other collection classes. However, this solution also has its drawbacks, and in particular it can cause problems if you iterate over a collection while another thread updates. This will result in an exception, and it is therefore necessary putting a lock by yourself. Therefore – and also for other reasons – there are defined some special thread safe collection classes:

1. *BlockingQueue* is a subinterface to *java.util.Queue* that implements blocking operations that wait for the queue is not empty before they return an item and do not add an item before the queue has room. The interface is implemented by the following classes: *ArrayBlockingQueue*, *LinkedBlockingQueue*, *DelayQueue*, *PriorityBlockingQueue* and *SynchronousQueue* and indirectly by *LinkedBlockingDeque* and *LinkedTransferQueue*.
2. *ConcurrentMap* is a subinterface to *java.util.Map* defining thread safe *putIfAbsent()*, *Remove()* and *replace()* methods. The interface is implemented by *ConcurrentHashMap*, *ConcurrentNavigableMap* and *ConcurrentSkipListMap*.

As an example is shown a program which solves the producer-consumer problem by means of a *BlockingQueue*:

```
package thread25;

import java.util.concurrent.*;

public class Thread25
{
 public static void main(String[] args)
 {
```

```
  final BlockingQueue<Character> queue = new ArrayBlockingQueue(26);
  final ExecutorService executor = Executors.newFixedThreadPool(2);
  final CountDownLatch done = new CountDownLatch(2);
  executor.execute(new Producer(queue, done));
  executor.execute(new Consumer(queue, done));
  try
  {
   done.await();
  }
  catch (InterruptedException ex)
  {
  }
  executor.shutdownNow();
 }
}

class Consumer implements Runnable
{
 private BlockingQueue<Character> queue;
 private CountDownLatch done;
```

```
 public Consumer(BlockingQueue<Character> queue, CountDownLatch done)
 {
  this.queue = queue;
  this.done = done;
 }

 public void run()
 {
  char ch = '\0';
  do
  {
   try
   {
    System.out.print(ch = queue.take());
   }
   catch (InterruptedException ie)
   {
   }
  }
  while (ch != 'Z');
  System.out.println();
  done.countDown();
 }
}

class Producer implements Runnable
{
 private BlockingQueue<Character> queue;
 private CountDownLatch done;

 public Producer(BlockingQueue<Character> queue, CountDownLatch done)
 {
  this.queue = queue;
  this.done = done;
 }

 public void run()
 {
  for (char ch = 'A'; ch <= 'Z'; ++ch)
  {
   try
   {
    queue.put(ch);
   }
```

```
    catch (InterruptedException ie)
    {
    }
   }
   done.countDown();
  }
 }
```

The code is self-explanatory, but you need to note how to create a *BlockingQueue* as an object of the type *ArrayBlockingQueue* with space for 26 elements. Also note that the program uses a *CountDownLatch* to control when the executor object can be shut down, so the program terminates.

The class *ConcurrentHashMap* is used basically in the same way as a *HashMap*, but it is extended with new thread safe methods, but without locking the entire data structure. One of these methods is *putIfAbsent()*, which adds a new key/value pairs, if there is not already a value with the same key. It is thus an alternative to the first performing *containsKey()*, and then *put()*, but such that the operation is indivisible. The method *remove()* works in much the same way, and there is also added a *replace()* method. In fact, the class expanded by more than 30 methods, including to provide support fort lambda expressions and aggregate operations. All these changes mean that a *ConcurrentHashMap* is suitable as a data structure into a cache memory.

EXERCISE 11

The subject of this exercise is to test the class *ConcurrentHashMap*. Create a new project, that you can call *MapProgram*. You must add the following class:

```
class Worker implements Runnable
{
 public static enum OPERATION { ADD, REP, REM }
 private final ConcurrentMap<String, BigInteger> map; // ConcurrentHashMap
 private final CountDownLatch done;       // when the thread is finish
 private String key;                      // key value, two upper case letters
 private OPERATION opr;                   // the operation, to be performed

 public Worker(ConcurrentMap<String, BigInteger> map, CountDownLatch done,
  String key, OPERATION opr)
 {
  this.map = map;
  this.done = done;
  this.key = key;
  this.opr = opr;
 }
```

```
 public void run()
 {
  // The method should using the method sum
  () determine a BigInteger for a
  // large random n
  // The method should then decided of opr add, change, or delete an item in
  // the map
  // finally, the method must print the result of the operation and
  // counting done down
 }

 private BigInteger sum(int n)
 {
  // determines and returns the sum of the numbers from 1 through n when the
  // sum is to be determined by add the numbers of the type BigInteger
  // meaning is that the operation must take time
 }
}
```

The program must then be written as follows:

```
public class MapProgram
{
 public static final Random rand = new Random();
 private static final int N = 2; // number of threads
 private static final int M = 5; // number of operations

 public static void main(String[] args)
 {
  CountDownLatch done = new CountDownLatch(M);
  ConcurrentMap<String, BigInteger> map = new ConcurrentHashMap();
  ExecutorService executor = Executors.newFixedThreadPool(N);
  for (int i = 0; i < M; ++i)
   executor.submit(new Worker(map, done, createKey(), operation()));
  // wait for done to be count down
  executor.shutdown();
 }

 private static Worker.OPERATION operation()
 {
  // returns a random operation
 }

 private static String createKey()
 {
  // creates a random key that is two upper case letters
 }

}
```

When you are finished you can experiment and explore what happens if you change the number of operations (the constant *M*), and if you change the number of concurrent threads (constant *N*). An example of running the program, which has 5 threads and 15 operations could be:

```
REM: TF is not found
REP: XN is not found
REP: NV is not found
REM: JG is not found
REP: GS is not found
REM: EM is not found
REP: BC is not found
ADD: DO sum(13340927) = 88990173280128
ADD: MA sum(12188689) = 74282075863705
```

```
REP: MC is not found
REP: LZ is not found
ADD: HW sum(11880594) = 70574262836715
REP: JP is not found
ADD: HK sum(14162771) = 100292048280606
REP: UJ is not found
```

8.10 PARALLELISM

The goal of multithreaded applications is parallelism, where several methods are performed at the same time on the machine, or at least it seems for us users as if it was the case when the individual threads in turn is running on the CPU in a very short period of time. That is the way, at least if the machine only has a single processor, but today most machines have more processors in the form of CPUs with multiple cores. Therefore such machines has the opportunity for true parallelism, where multiple threads run at the same time, each on their core. The basic primitives for threads, and also the concurrency tools discussed above, however, are not very effective to use multiple cores or processors. For it really to be possible, the task must be parallels and thus could be divided into smaller tasks that can be performed independent of each other and to finish be assembling to the finished result. This is far from everything, but there are also tasks that can be easily parallels. To support that Java has a framework, called the *Fork/Join* that exactly supports the resolution of these kinds of tasks.

Fork/Join consists of a special executor service and a thread pool, and each task is of the framework split into smaller pieces that are forked to a thread from the thread pool. A task waits until the subtasks are completed and result of each sub-task can be merged (joined) for the final result. This process is also recursive, where subtasks in turn can be divided into subtasks. The framework basically consists of the following classes:

1. *ForkJoinPool*
2. *ForkJoinTask*
3. *ForkJoinWorkerThread*
4. *RecursiveAction*
5. *RecursiveTask*
6. *CountedCompleter.*

Consider as an example the following program:

```
package thread26;

public class Thread26
{
 private static int[][] A1 = { { 1, 2, 3 }, { 4, 5, 6} };
 private static int[][] A2 = { { 1, 2, 3 }, { 4, 5, 6}, { 7, 8, 9 } };

 public static void main(String[] args)
 {
  Matrix A = new Matrix(A1);
  print(A);
  Matrix B = new Matrix(A2);
  print(B);
  print(mul(A, B));
}

 public static Matrix mul(Matrix A, Matrix B)
 {
  if (A.getCols() != B.getRows())
   throw new IllegalArgumentException("Ulovlig matrixmultiplikation");
  Matrix C = new Matrix(A.getRows(), B.getCols());
```

```
 for (int i = 0; i < A.getRows(); ++i)
  for (int j = 0; j < B.getCols(); ++j)
   for (int k = 0; k < A.getCols(); ++k)
    C.setValue(i, j, C.getValue(i, j) + A.getValue(i, k) * B.getValue(k, j));
 return C;
}

public static void print(Matrix M)
{
 for (int i = 0; i < M.getRows(); ++i)
 {
  for (int j = 0; j < M.getCols(); j++)
   System.out.printf("%5d", M.getValue(i, j));
  System.out.println();
 }
 System.out.println();
}


lass Matrix

private final int[][] matrix;

public Matrix(int rows, int cols)
{
 matrix = new int[rows][cols];
}

public Matrix(int[][] array)
{
 matrix = array;
}

public int getCols()
{
 return matrix[0].length;
}

public int getRows()
{
 return matrix.length;
}

public int getValue(int row, int col)
{
 return matrix[row][col];
}
```

```
 public void setValue(int row, int col, int value)
 {
  matrix[row][col] = value;
 }
}
```

Here is the class *Matrix* a simple matrix class (with *int* coordinates), and it is not much more than a thin encapsulation of a 2 dimensional array. The program creates two matrices and prints them. One is a 2×3 matrix, and the second is a 3×3 matrix. Therefore, you can compute the two matrices product (the first matrix has the same number of columns as the second matrix has rows), and that's exactly what method mul() does. If you executes the program, you get the following result:

```
     1     2     3
     4     5     6

     1     2     3
     4     5     6
     7     8     9

    30    36    42
    66    81    96
```

The program is carried out in a single thread (the primay thread), but if you consider the method *mul()*, then you can also write it as follows:

```
public static Matrix mul(Matrix A, Matrix B)
{
 if (A.getCols() != B.getRows())
  throw new IllegalArgumentException("Ulovlig matrixmultiplikation");
 Matrix C = new Matrix(A.getRows(), B.getCols());
 for (int i = 0; i < A.getRows(); ++i) mul(A, B, C, i);
 return C;
}
public static void mul(Matrix A, Matrix B, Matrix C, int r)
{
 for (int j = 0; j < B.getCols(); ++j)
  for (int k = 0; k < A.getCols(); ++k)
   C.setValue(r, j, C.getValue(r, j) + A.getValue(r, k) * B.getValue(k, j));
}
```

This means that the matrix multiplication is divided into sub-tasks, where there is a subtask for each row in the first matrix. Each of these sub-tasks are actually independent of each other and can therefore be performed in separate thread, and if you can maintain that the threads are executed on their own processor, one would expect an improved efficiency. Not in this case, where there are few rows and columns, but for large matrices, the matrix multiplication is actually an extensive operation with a bad time complexity.

The task is thus an example of a job where you can experiment with *Fork/Join* what the following program does (I have not shown the class Matrix, as it is the same as above):

```
package thread27;

import java.util.*;
import java.util.concurrent.*;

public class Thread27 extends RecursiveAction
{
 private static int[][] A1 = { { 1, 2, 3 }, { 4, 5, 6} };
 private static int[][] A2 = { { 1, 2, 3 }, { 4, 5, 6}, { 7, 8, 9 } };
```

```
public static void main(String[] args)
{
 Matrix A = new Matrix(A1);
 print(A);
 Matrix B = new Matrix(A2);
 print(B);
 Matrix C = new Matrix(A.getRows(), B.getCols());
 ForkJoinPool pool = new ForkJoinPool();
 pool.invoke(new Thread27(A, B, C));
 print(C);
}

private final Matrix A, B, C;
private final int row;

public Thread27(Matrix A, Matrix B, Matrix C)
{
 this(A, B, C, -1);
}

public Thread27(Matrix A, Matrix B, Matrix C, int row)
{
 if (A.getCols() != B.getRows())
  throw new IllegalArgumentException("Ulovlig matrixmultiplikation");
 this.A = A;
 this.B = B;
 this.C = C;
 this.row = row;
}

public void compute()
{
 if (row == -1)
 {
  List<Thread27> tasks = new ArrayList();
  for (int r = 0; r < A.getRows(); ++r) tasks.add(new Thread27(A, B, C, r));
  invokeAll(tasks);
 }
 else mul(A, B, C, row);
}

public static void mul(Matrix A, Matrix B, Matrix C, int r)
{
 for (int j = 0; j < B.getCols(); ++j)
  for (int k = 0; k < A.getCols(); ++k)
   C.setValue(r, j, C.getValue(r, j) + A.getValue(r, k) * B.getValue(k, j));
}
```

```
 public static void print(Matrix M)
 {
  for (int i = 0; i < M.getRows(); ++i)
  {
   for (int j = 0; j < M.getCols(); j++)
     System.out.printf("%5d", M.getValue(i, j));
   System.out.println();
  }
  System.out.println();
 }
}
```

The program is technical, and the following points only at the most central. If a task – a class – must be parallels, it must inherits the class *RecursiveAction*, which is the class that contains the whole secret. The class is abstract and has an abstract method *compute()*, and it is there that the interesting thing is, seen from the programmer. The class has two constructors, that is used, respectively by three matrices, and three matrices and a row number. The constructors do nothing but initialize the corresponding instance variables. If you look at the *main()* method, it starts to create the matrices, but then it creates a thread pool to the *Fork/Join*:

```
ForkJoinPool pool = new ForkJoinPool();
```

It is subsequently used to create a *RecursiveAction* object (the task) and assign it to the pool object:

```
pool.invoke(new Thread27(A, B, C));
```

The result of this is that the method *compute()* is performed. If the instance variable *row* has the value -1 (and it has the first time), then the task is split into a list of tasks, which parallels with the statement

```
invokeAll(tasks);
```

If the *row* not is -1, then instead the method *mul()* is called, which performs the matrix multiplication for a specific row. If the program is performed, you get the same result as above.

EXERCISE 12

Create a new project, that you can call *Matrices1*. The project should be substantially a copy of the project *Thread26*, and thus a program that can calculate the product of two matrices, but it should this time be larger matrices. Start by adding a method that can create a matrix with a given number of rows and columns and initialize the matrix with random one-digit integers. You can then test the program with 1000×2000 and 2000×1000 matrices. You can not print the matrices (that do not really make sense with such large matrices), but you should instead measure and print how long time the matrix multiplication takes. Remember only to measure the time for the matrix multiplication.

You must then create a similar project that you can call *Matrices2*, but this time the program should work in the same way as the program *Thread27* where the matrix multiplication should be parallels. Try to observe a time difference between the two programs. That should be able, if your machine has a multicore processor.

8.11 *COMPLETIONSERVICE*

As the last synchronization tool I'll mention a *CompletionService*, which is an interface that defines a service that detaches the creation of an asynchronous tasks from the handling of the result. The idea is that a producer uses *submit()* to start callable objects in a thread, and a consumer can call a blocking *take()* method that is waiting for the result.

```
package thread28;

import java.math.*;
import java.util.concurrent.*;

public class Thread28
{
 private static final int N = 10;

 public static void main(String[] args) throws Exception
 {
  ExecutorService executor = Executors.newFixedThreadPool(N);
  CompletionService<BigDecimal> completion =
   new ExecutorCompletionService<BigDecimal>(executor);
  Future<BigDecimal>[] res = new Future[N];
  for (int i = 0, n = 100; i < N; ++i, n += 100)
  {
   res[i] = completion.submit(new ECalculator(n));
   completion.take();
  }
  for (int i = 0; i < res.length; ++i) System.out.println(res[i].get());
  executor.shutdown();
 }
}

class ECalculator implements Callable<BigDecimal>
{
 private final int dec;

 public ECalculator(int dec)
 {
  this.dec = dec;
 }

 public BigDecimal call()
 {
  MathContext mc = new MathContext(dec, RoundingMode.HALF_UP);
  BigDecimal y = BigDecimal.ZERO;
```

```
  for (int i = 0; ; ++i)
  {
   BigDecimal fac = BigDecimal.ONE.divide(factorial(new BigDecimal(i)), mc);
   BigDecimal z = y.add(fac, mc);
   if (z.compareTo(y) == 0) break;
   y = z;
  }
  System.out.println("Finish");
  return y;
 }

 private BigDecimal factorial(BigDecimal n)
 {
  return n.equals(BigDecimal.ZERO) ?
   BigDecimal.ONE : n.multiply(factorial(n. subtract(BigDecimal.ONE)));
 }
}
```

In the above program, the class *ECalculator* is the same as I have used before – with only one difference that the method *call()* prints a text before it returns the result. The goal is to show that all threads are performed before the result is printed. In *main()* is created as usual an *ExecutorService* object, but it is encapsulated in a *CompletionService* object. Then are started 10 threads on basis of a *Callable* object so that each thread determines the number *e* with a number of decimal places (a time-consuming operation), and for each thread is performed the statement:

```
completion.take();
```

This means that the primary thread waits until the 10 worker threads are completed.

9 ATOMIC VARIABLER

Finally, I will mention some classes that can be used to ensure mutual exclusion of operations on simple variables, but without the need to ensure the code using locks which generally cost performance. In fact, several simple operations on variables of primitive types is not thread safe, but consist of more operations and thus can be interrupted. The classes are in

java.util.concurrent.atomic

and the main classes are

- *AtomicBoolean*
- *AtomicInteger*
- *AtomicIntegerArray*
- *AtomicLong*
- *AtomicLongArray*
- *AtomicReference*
- *AtomicReferenceArray*
- *DoubleAccumulator*

- *DoubleAdder*
- *LongAccumulator*
- *LongAdder*

If, for example you need to implement a class that generates a continuous id, you will typically write something like the following:

```
class ID
{
 private static volatile long ID = 1;

 public static synchronized long getID()
 {
  return ID++;
 }
}
```

where the method *getID()* is *synchronized*, since the ++ operation is not thread safe. Such a solution works and will also in most contexts be fine enough, but every time you need a new id, you set a lock and that is not free. However, you can write the class as follows:

```
import java.util.concurrent.atomic.*;

class ID
{
 private static AtomicLong ID = new AtomicLong(1);

 static long getID()
 {
  return ID.getAndIncrement();
 }
}
```

Here are *getID()* also thread safe, but the method is significantly more efficient, as there is not set a lock. *AtomicLong* is a class that encapsulates a *long* and has a number of methods that perform the usual operations on a long, but such they are thread safe.

10 SWING

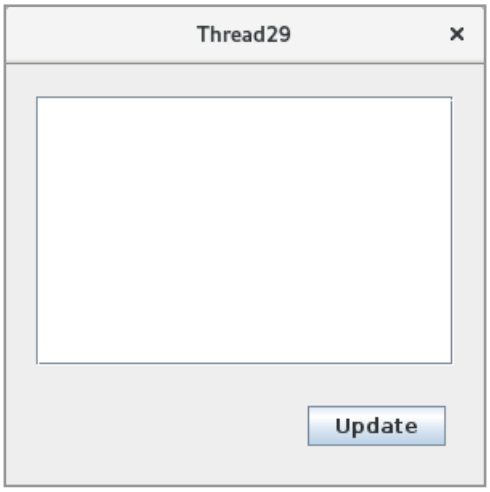
If you have a GUI application, threads generally can create problems, where the user interface is not updated correctly, and maybe you even get an exception. In this section I will show how to use threads within an application written using swing.

Swing is single-threaded, and if you do nothing else, what is the case in all the GUI programs in this book until this place, where it happens all in a single thread. The thread that renders the graphics (windows and components), that also deal with events, is called the *event dispatcher thread* and is commonly referred to as EDT. This thread processes all events that come from the underlying event queue and calls the component's event handlers, so they are performed by EDT. This means that everything regarding drawing in the window must takes place in this thread. This also means that to be sure that a Swing program works correctly, you have to be careful how the program's code interacts with the EDT, and especially pay attention to

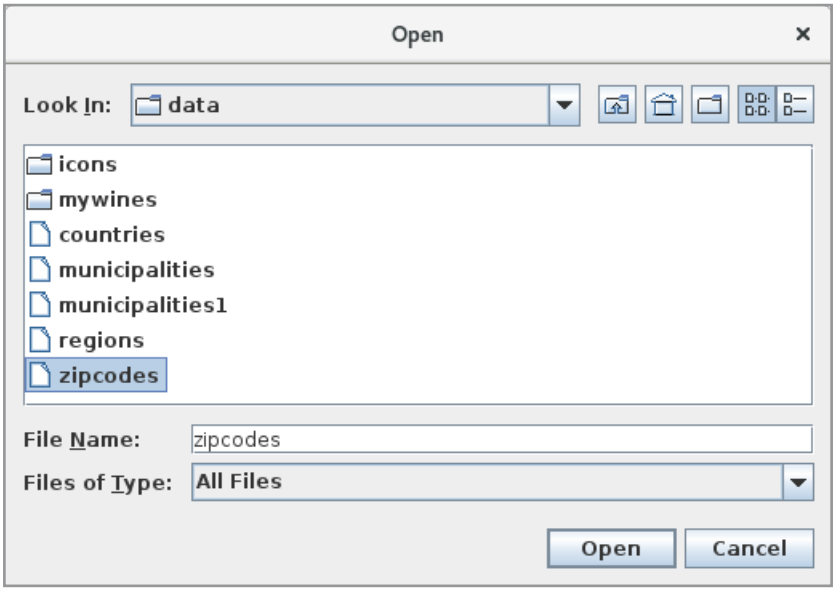
1. that it is always EDT, that creates GUI components
2. that you not must perform code that slows EDT

Since Swing is single-threaded, should a Swing program always create the GUI part in the EDT thread and never create components in another thread, nor in the primary thread. Most Swing components and including *JFrame* is not thread safe, and use these components from a different thread than EDT, there is a risk that the user interface is not updated correctly.

To explain a little of all that, I will show a program that will open the following window, which has a list box and a button:



If you click the button, you get a dialog box where you have to select the file with zip codes (which I have used several times):



and opens this file the zip codes appears in the main window's list box. The code for the main window are as follows:

```
package thread29;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;

public class MainView extends JFrame
{
 private DefaultListModel model = new DefaultListModel();
 private JButton cmd = new JButton("Opdater");

 public MainView()
 {
  super("Thread29");
  setSize(300, 300);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  createView();
  setVisible(true);
 }

 private void createView()
 {
  JPanel bottom = new JPanel(new FlowLayout(FlowLayout.RIGHT));
  cmd.addActionListener(this::update);
  bottom.add(cmd);
  JPanel panel = new JPanel(new BorderLayout(0, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  panel.add(new JScrollPane(new JList(model)));
  panel.add(bottom, BorderLayout.SOUTH);
  add(panel);
  JOptionPane.showMessageDialog(this, Thread.currentThread().getId());
 }

 public void update(ActionEvent e)
 {
  JOptionPane.showMessageDialog(this, Thread.currentThread().getId());
  final JFileChooser chooser = new JFileChooser();
  if (chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
  {
```

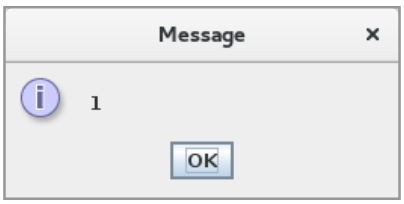
```
    update(chooser.getSelectedFile());
    cmd.setEnabled(false);
   }
  }

  private void update(File file)
  {
   try
   {
    BufferedReader reader = new BufferedReader(new FileReader(file));
    for (String line = reader.readLine(); line != null; line = reader.readLine())
    {
     String[] elem = line.split(";");
     if (elem.length == 2) model.addElement(elem[0] + " " + elem[1]);
     busy();
    }
    reader.close();
   }
   catch (Exception ex)
   {
    model.addElement(ex.toString());
   }
  }

  private void busy()
  {
   for (int i = 0; i < 500000; ++i) Math.cos(Math.sqrt(Math.PI));
  }
}
```

and it contains nothing new, but a couple of the statements require an explanation. The last method *busy()* is a dummy method and is a method that requires some time. It is used in the method *update()*, which is the method that reads the file with zip codes where it is carried out every time there is read a line in the file. The consequence is that it takes a long time to update the list box. The method *update()*, which is called from the event handler for the button, that start by displaying a message box that shows the id of the current thread. Similarly, the last statement in the method *createView()* displays a message box that shows the current thread id.

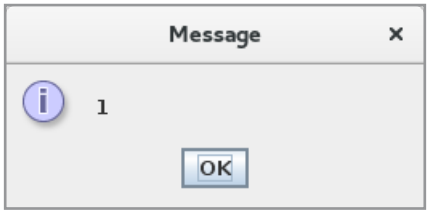
The *main()* method is the following:

```
package thread29;

public class Thread29
{
 public static void main(String[] args)
 {
  javax.swing.JOptionPane.showMessageDialog(null,
   Thread.currentThread().getId());
  new MainView();
 }
}
```

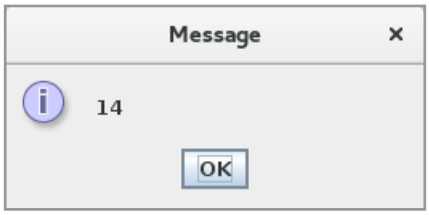
and here is nothing new beyond that it also starts to show a message box with the id of the current thread. If you now run the program, you first get a message box:



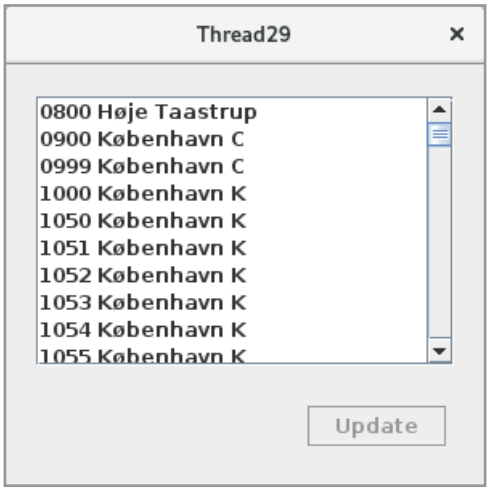
which opens in the *main()* method. It tells that the message box is opened in the primary thread (which was also to be expected). If you here click OK, you get the message box again:



but this time the message box is opened in the method *createView()* in the class *MainView*, and it is still happened from the primary thread. That is that the window is created in the primary thread. When you here click OK, the main window opens. Clicking on the button, you get the following message box:



which is opened in the button's event handler, and you can see that it happens in a different thread. This is the event dispatcher thread, and if you click OK, you get chance to browse the file, after which the list box is updated:



After you've accepted the file, it takes a relatively long time before the list box is updated, and in that time the program does not respond. You can change the window size, but you will find that the window is not redrawn. It happens only when the event handler is completed. The program has – at least – two problems. First, it is carried out the code that creates the window and its components, and the event handler in the two different threads. As long as the program does not use other threads, it will probably not cause problems, but it is not consistent with how a Swing program should be written. Second, the program has a time when it does not respond to user actions (here obviously unnecessarily long because of the method *busy()*), but overall it is a requirement that a GUI application must react sensibly to the user's interaction with the program.

I would start by solving the first problem, and in the project *Thread30* the main program is the following:

```
package thread30;

public class Thread30
{
 public static void main(String[] args)
 {
  javax.swing.SwingUtilities.invokeLater(() -> new MainView());
 }
}
```

This means that the window is now created by the event dispatcher thread – the EDT thread. The class *MainView* is unchanged (except another text in the title bar), and if you run the program, you will see that the window is created in the same thread that performs the event handler.

The class *SwingUtilities* has a number of static methods that are used in a variety of contexts in Swing. For threads these are basically of three methods:

1. *invokeAndWait()* that has a *Runnable* objet as a parameter and executes the current *run()* method in the thread EDT. The method is carried out synchronism with EDT, and *invokeAndWait()* blocks the current thread until the EDT has processed all pending events. *invokeAndWait()* is used when the program must update the UI from a different thread than EDT. This method should never be called from EDT.

2. *invokeLater()* also has a *Runnable* object as a parameter, and performs the *run()* method in EDT, but it happens asynchronously this time, and the *run()* method is performed only after the EDT has processed pending events. *invokeLater()* is used when an application wants to update the user interface, and the method can be called from any thread.
3. *isEventDispatchThread()* is a method that returns *true*, if it is performed in EDT.

Looking at the above, the *main()* method defines (by means of a lambda expression) a *Runnable* object to instantiate the main window. The object is used as a parameter to *invokeLater()*, and the result is that it is the event dispatcher thread, which creates the window. Similar to the above, it means that all the GUI programs that I so far have shown are not written correctly, and although the programs have hardly any significance, I will continue writing GUI applications as shown in the *main()* method above.

Then there's the other problem with the program is not responding while it loads the zip codes. It is clear that the problem must be solved by loading the zip codes in a worker thread, but there is a little more, because you can be sure that the user interface is updated correctly. The problem is solved in the project *Thread31*, and the only thing that has changed are the following two methods:

```
public void opdater(ActionEvent e)
{
 final JFileChooser chooser = new JFileChooser();
 if (chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
 {
  new Thread(() -> update(chooser.getSelectedFile())).start();
  cmd.setEnabled(false);
 }
}

private void update(File file)
{
 try
 {
  BufferedReader reader = new BufferedReader(new FileReader(file));
  for (String line = reader.readLine(); line != null; line = reader.readLine())
  {
   String[] elem = line.split(";");
   if (elem.length == 2) SwingUtilities.invokeAndWait(
     () -> model.addElement(elem[0] + " " + elem[1]));
   busy();
  }
  reader.close();
 }
 catch (Exception ex)
 {
  model.addElement(ex.toString());
 }
}
```

In the event handler is now started a thread which performs the method *update()*. This method is largely unchanged, but when the list box need to be updated, it must according to the above be in EDT, and when the update was carried out in a runnable object it is performed by *invokeAndWait()* and thus in the thread EDT. In this case it used *invokeAndWait()* instead of *invokeLater()*, but it has hardly any meaning, but *invokeAndWait()* ensures that no events concerning something else are hanging.

If you try out the program *Thread31* you will find that the program now "is in live" while the list box is updated.

10.1 SWINGWORKER

Sometimes there is a need to start a task that takes time (for example to load the zip codes from a file), and to reduce the number of updates to the user interface, and only update the user interface at once, after the job is done. You can then use a class called *SwingWorker*. It is a generic class, as is parameterized with two parameters *SwingWorker<T, V>*. It is an abstract class, and there is at least one method to be overridden. The first is

protected abstract T doInBackground()

which is the method that performs the task for a worker thread, which returns a value of the first parameter type (if there is no return, specify the parameter type as *Void*). When this method terminates, the following method is performed:

protected void done()

but such that it is performed in EDT, where you can safely update the user interface. The default implemntation performs nothing, and you will often override the method. While the task is performed, you can, if desired periodically update the user interface by overriding and call the method

protected void process(List<V> chunks)

The class *SwingWorker* have additional two methods that you need to know:

1. *execute()* starts a worker thread
2. *T get()* waits to *doInBackground()* is terminated and returns the result

Below is how the program with the zip code must be changed if the list box should be updated with a *SwingWorker*:

```
package thread32;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;
import java.util.*;
```

```
public class MainView extends JFrame
{
 private DefaultListModel model = new DefaultListModel();
 private JButton cmd = new JButton("Update");

 public MainView()
 {
  super("Thread32");
  setSize(300, 300);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  createView();
  setVisible(true);
 }

 private void createView()
 {
  JPanel bottom = new JPanel(new FlowLayout(FlowLayout.RIGHT));
  cmd.addActionListener(this::update);
  bottom.add(cmd);
  JPanel panel = new JPanel(new BorderLayout(0, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  panel.add(new JScrollPane(new JList(model)));
```

```
 panel.add(bottom, BorderLayout.SOUTH);
 add(panel);
}

public void update(ActionEvent e)
{
 final JFileChooser chooser = new JFileChooser();
 if (chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
 {
  new Zipcodes(chooser.getSelectedFile()).execute();
  cmd.setEnabled(false);
 }
}

class Zipcodes extends SwingWorker<ArrayList<String>, Void>
{
 private final File file;

 public Zipcodes(File file)
 {
  this.file = file;
 }

 public ArrayList<String> doInBackground()
 {
  ArrayList<String> list = new ArrayList();
  try
  {
   BufferedReader reader = new BufferedReader(new FileReader(file));
   for (String line = reader.readLine(); line != null;
    line = reader.readLine())
   {
    String[] elem = line.split(";");
    if (elem.length == 2) list.add(elem[0] + " " + elem[1]);
   }
   reader.close();
  }
  catch (Exception ex)
  {
   list.clear();
  }
  return list;
 }
```

```
 public void done()
 {
  try
  {
   ArrayList<String> list = get();
   for (String line : list) model.addElement(line);
  }
  catch (Exception ex)
  {
   model.clear();
  }
 }
}
```

The important thing is the class *Zipcodes*, that is an inner class. It inherits the class

```
SwingWorker<ArrayList<String>, Void>
```

parameterized by a single parameter (the last should not to be used), and the class must therefore implement the method *doInBackground()*, which is the code that performs the worker thread (a different thread than EDT). The method reads the contents of the file and saves it as lines in a *ArrayList<String>*, which then is the value that the method returns. Note that I have removed the method *busy()* – it is no longer necessary to illustrate that it takes time to read the zip codes. The class overrides also the method *done()*, and it is performed by EDT after *doInBackground()* is executed. Note how to get the return value with the method *get()*, and how the value is used to update the list box.

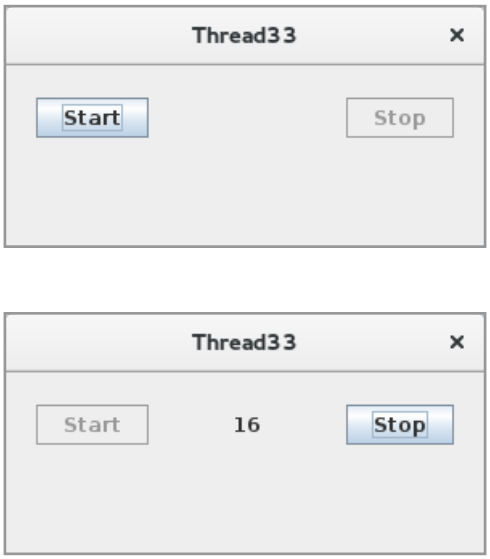
Finally, there is to note how the worker thread starts in the event handler:

```
new Zipcodes(chooser.getSelectedFile()).execute();
```

10.2 A TIMER

I have previously shown how to use a *Timer*, but Swing also defines a *Timer* class that in the same manner as previously performs a particular method at certain times, but the main difference and its justification is that it is performed in EDT. It works as, after an initial period it at a certain time interval fires an *ActionEvent*, which registered listeners can catch.

The program *Thread33* opens a window as shown below, where there are two buttons. Between the two buttons is a *JLabel*, which from the start is blank. If you click on the *Start* button, the program starts a timer that ticks every second where a *counter* variable is increased by 1 and the value is shown in the label component. If you click the *Stop* button the timer stops, and you can start it again by clicking *Start*.



The code is the following:

```
package thread33;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class MainView extends JFrame
{
 private DefaultListModel model = new DefaultListModel();
 private JButton cmd1 = new JButton("Start");
 private JButton cmd2 = new JButton("Stop");
 private JLabel lbl = new JLabel();
 private Timer timer = null;
 private int counter = 0;

 public MainView()
 {
  super("Thread33");
  setSize(300, 150);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  createView();
  setVisible(true);
 }

 private void createView()
 {
  JPanel top = new JPanel(new BorderLayout(20, 0));
  cmd1.addActionListener(this::start);
  cmd2.addActionListener(this::stop);
  cmd2.setEnabled(false);
  top.add(cmd1, BorderLayout.WEST);
  top.add(cmd2, BorderLayout.EAST);
  top.add(lbl);
  JPanel panel = new JPanel(new BorderLayout(0, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  panel.add(top, BorderLayout.NORTH);
  add(panel);
  timer = new Timer(1000, this::tick);
  lbl.setHorizontalAlignment(JLabel.CENTER);
 }
```

```
 public void tick(ActionEvent e)
 {
  lbl.setText("" + (++counter));
 }

 public void start(ActionEvent e)
 {
  timer.start();
  cmd1.setEnabled(false);
  cmd2.setEnabled(true);
 }

 public void stop(ActionEvent e)
 {
  timer.stop();
  cmd1.setEnabled(true);
  cmd2.setEnabled(false);
 }
}
```

There is not much to explain, but you should note how to create a timer and assigns an event handler.

11 CALENDAR

As a final example, I will show the development of a program that can display a calendar on the screen, but in addition, the program should keep track of appointments and the like. There are many such programs for any platform, and the goal of the following is primarily to write a program that uses multiple threads. When you study the finished program code or test the program should also notice that the application uses more details on Swing, not mentioned in the previous books.

11.1 TASK FORMULATION

The task is to write a program that in a window can display a calendar. The window should show the calendar for a month and it should be possible to navigate in the calendar, for example next month, next year and so on. It should also be possible to enter a specific year and a month and then go directly to that month. The program shall cover the period from year 0 to year 9999, and the program must take account of the shift from the Julian calendar to the Gregorian calendar. The program should also be able to save a calendar for a selected period to a text file.

The program should be used as a daily calendar program, and one should therefore also be able to enter notes and appointments, set alarms and set special (custom) anniversaries beyond the days that the calendar was born with (holidays). The calendar must give a warning, when the time for an appointments occurs.

Remark

The result of this first phase is a project library, as preliminary only have a subdirectory with this task formulation.

11.2 ANALYSIS

The analysis will consist of

1. a requirement specification
2. a prototype for the user interface

11.2.1 REQUIREMENT SPECIFICATION

The main application window should primarily show a calendar for a specific month.

The calendar will cover the period from year 0 to year 9999 and should take account of the shift from the Julian to the Gregorian calendar. It is decided that the switch from the Julian to the Gregorian calendar is set to 1582, and then the 4 October is followed by the 15 October.

For each date, the calendar must display the following information:

- the day's number in the month
- the week day's name
- the week's number in the year, if it is a monday
- an indication if it is a public holiday or anniversary
- a mark when created notes
- a mark when created appointments
- a mark when created anniversaries

For the holidays, the program must be pre-programmed to (know) the following days:

- New Year's Day
- Palm Sunday
- Maundy Thursday
- Good Friday
- Easter Sunday
- Easter Monday
- Prayer Day
- Christ's Ascension
- Pentecost
- Whit Monday
- Christmas Eve
- Christmas Day
- Second Christmas Day

and also the program must be able to show custom anniversaries (for example the wife's birthday). An anniversary falls on a specific date and every year after 1582 (with the option to specify a start and end year). If an anniversary specifies d. 29/2, and it is not a leap year the date ⅓ should be used.

The program distinguishes between notes and appointments. A note is any text entered for a specific date, and there can be more notes for the same date. An appointment is a short message relating to a specific time (clock) and the program must be able to automatically notifies the user when the time for an appointment occurs. Notes will be saved until the user manult delete them. Appointments should be automatically deleted when the time is exceeded, but maybe there should be an option that an appointment should be stored after the time is exceeded.

Regarding appointments it must be able to make an appointment for a specific date and for a specific time. There may then be several appointments the same day, and an appointment is in principle only plain text. An appointment can span multiple days. An appointment can also have a start time, an end time, or both, and if so, the program should come with a warning if the timing conflicts. When the time of an appointment occurs, the program must come with a warning that must appear in the user interface. It is adopted

1. the warning should appear at the day's start (when the computer is turned on)
2. 1 hour before the appointment occurs
3. 15 minutes before the appointment occurs
4. 5 minutes before the appointment occurs
5. when the appointment occurs, and the appointment should be deleted

Functions:

Navigate the calendar:

- Shift to previous year
- Shift to previous month
- Shift to next month
- Shift to next year
- Enter year and month and the calendar must shift to that month

The program must be able to save a calendar as a comma delimited text file with the following format:

```
year; month number; month name; day in month; day name; week number [; holiday]
```

and an example could be

```
2012;4;April;1;Sunday;14;Palm Sunday
2012;4;April;2;Monday;14;
2012;4;April;3;Tuesday;14;
```

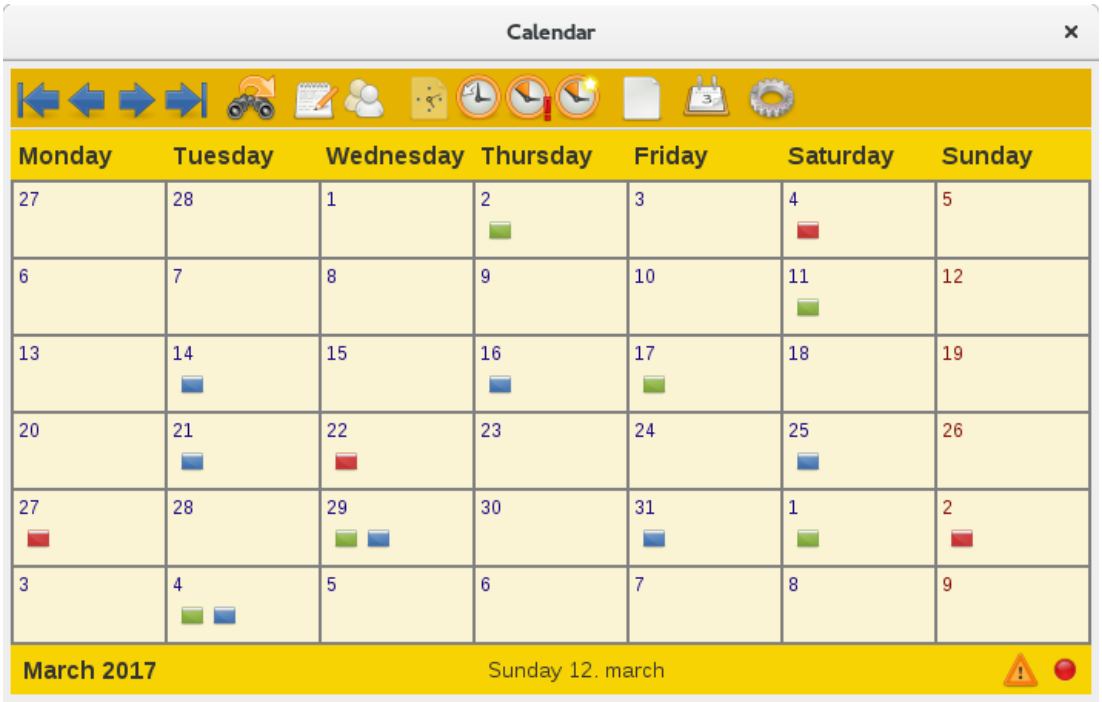
The program must also have the following features:

- Maintenance of anniversaries
- Maintenance of notes
- Maintenance of appointments
- Watch
- Alarm (that comes with a warning a certain time)
- Timer (that comes with a warning when a timer reach 0)
- Stopwatch

11.2.2 THE PROTOTYPE

The prototype is a NetBeans project called *Calendar*. The result is a program that only opens the window below. All the program's functions are placed in a toolbar at the top of the window. At the bottom is a status bar showing

- the current month
- the current day (today)
- an icon that shows whether there is set an alarm (if not, no icon appears)
- an icon that shows whether there is set a timer (if not, no icon appears)



For each date is shown the day's number in the month (and later the week number), and the color indicates whether it is a public holiday. In addition, there are three icons, which means that there are

1. notes for that date
2. appointments for that date
3. anniversaries of that date

If you click on an icon, you get a simple dialog box with information, and for appointments it should also be possible to cancel the appointment.

If you right click on a specific date you get a menu where you can

1. create a note
2. create an appointment
3. create an anniversary

As for the icons in the toolbar, the following applies:

1. The first four icons (arrows) are used to navigate the calendar.
2. The next opens a dialog box so you can navigate to a specific month.
3. The sixth icon opens a dialog to maintenance notes.

4. The seventh icon opens a dialog to maintenance appointments.
5. The next shows the clock (how the clock should look like is not yet determined).
6. The next again is for the stopwatch. Opens a dialog box to start or stop the stopwatch.
7. The tenth icon opens a dialog box to maintenance of alarms.
8. The eleventh icon opens a dialog box to maintaining timers.
9. The next icon is to export of a calendar to a textfile and opens a dialog box for selecting the period.
10. The second last icon opens a dialog box to maintenance of anniversaries.
11. The last icon is the settings for the program, but it has not yet determined which settings should be talking about, but possibly colors.

11.3 DESIGN

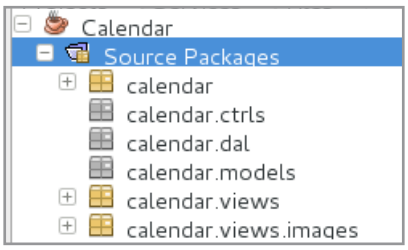
The design phase are performed in five steps:

1. Design of the architecture
2. Design of the model layer
3. Design of the DAL layer
4. Design of the user interface
5. Design of the controller layer

The design starts to create a copy of NetBeans project from the analysis (the prototype). The copy is called *Calendar1*.

11.3.1 DESIGN OF THE ARCHITECTURE

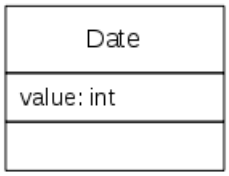
The program's architecture has a classic MVC architecture:



The emphasis of the project is the model layer that has relatively complex algorithms.

11.3.2 DESIGN OF THE MODEL LAYER

Java has classes that represent a date, but since it is a requirement that the calendar must support the change from the Julian calendar to the Gregorian calendar, it is decided to implement a custom date type. The class must represent a date as an *int* in the format YYYYMMDD:



The class must have a number of methods, and the following design shows only the most important, but there will be many more:

```
package calendar.models;

/**
 * Represents a date between year 0 and year 9999. The class takes into
 * account the shift from the Julian to the Gregorian calendar.
 */
```

```
public class Date implements Comparable<Date>
{
 private int value; // represents a date as YYYYMMDD

 /**
  * Returns the days number int the week :
  * 1 = monday
  * 2 = tuesday
  * 3 = wednesday
  * 4 = thursday
  * 5 = friday
  * 6 = saturday
  * 7 = sunday
  * @return The days number int the week
  */
 public int getWeekDay()
 {
  throw new UnsupportedOperationException();
 }

 /**
  * Moves the current date, a day ahead.
  */
 public void nextDay()
 {
  throw new UnsupportedOperationException();
 }

 /**
  * Moves the current date back a day.
  */
 public void prevDay()
 {
  throw new UnsupportedOperationException();
 }

 /**
  * Moves the current date, a month ahead.
  */
 public void nextMonth()
 {
  throw new UnsupportedOperationException();
 }

 /**
  * Moves the current date back a month.
  */
```

```
public void prevMonth()
{
 throw new UnsupportedOperationException();
}

/**
 * Moves the current date, a year ahead.
 */
public void nextYear()
{
 throw new UnsupportedOperationException();
}

/**
 * Moves the current date back a year.
 */
public void prevYear()
{
 throw new UnsupportedOperationException();
}

/**
 * Calculate the number of the week in the year, where week number 1
 * is the first week in the year that contains a thursday.
 * @return the number of the week in the year
 */
public int getWeekNumber()
{
 throw new UnsupportedOperationException();
}

/**
 * If this date is a holiday the method returns the name, and else the
 * method returns null.
 * The method can return the following names:
 * New Year's Day
 * Palm Sunday
 * Maundy Thursday
 * Good Friday
 * Easter Sunday
 * Easter Monday
 * Prayer Day
 * Christ's Ascension
 * Pentecost
 * Whit Monday
 * Christmas Eve
```

```
   * Christmas Day
   * Second Christmas Day
   * @return the name of a holiday or null
   */
  public String getHoliday()
  {
   throw new UnsupportedOperationException();
  }

  public int compareTo(Date date)
  {
   throw new UnsupportedOperationException();
  }
}
```

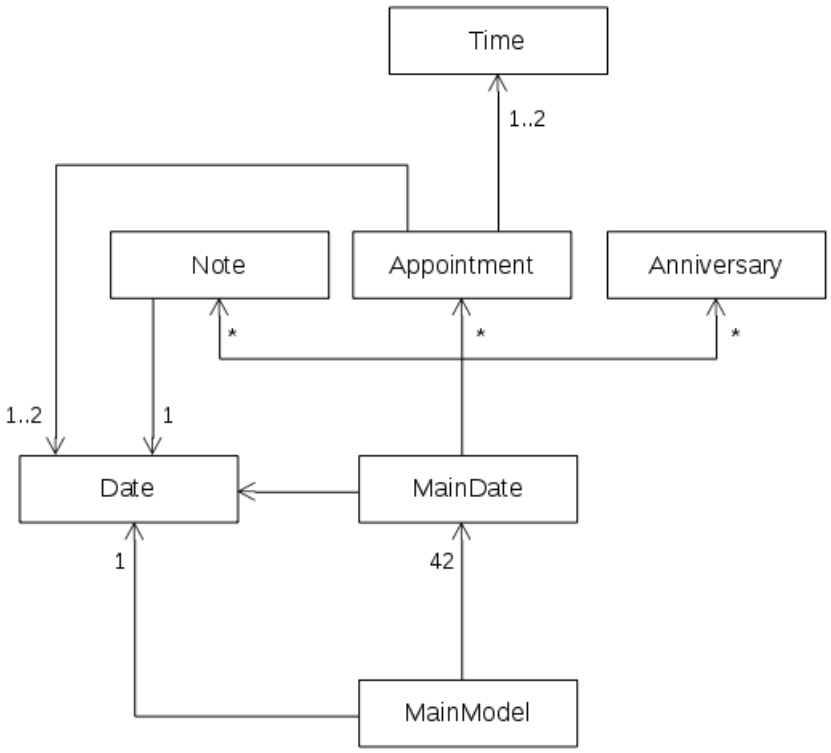
The method *getWeekDay()* is complex because it must return the correct day for the entire range of dates, that the calendar span. The approach is to implement an algorithm that determines the number of days from d. 1/1 year 0 and up to a given date. If you determines the week day for a known date, then the day of the week can be calculated.

The calculation methods for moving the calendar one day, month or year forward or backward are also all relatively complex because they must take into account leap years and the shift between the Julian and Gregorian calendar.

The method *getWeekNumber()* is implemented using a method which determines the number of days from the beginning of the year to the current date.

Finally, there is the method *getHoliday()* which is a complex algorithm. Some of the public holidays fall on fixed dates and are unproblematic, while the other can be determined from the date of Easter Sunday. There are algorithms that can determine Easter Sunday, and the method can therefore be implemented by implement such an algorithm. I will use an algorithm called Gaus's algorithm.

Based in the class *Date*, the program's data model is outlined as follows:



The other classes in the data model are defined as follows:

```
package calendar.models;

/**
 * Represents a time.
 */
```

```
public class Time implements Comparable<Time>
{
 private int value; // represents a time as HHMMSS

 public int compareTo(Time time)
 {
  throw new UnsupportedOperationException();
 }
}
```

```
package calendar.models;

public class Appointment
{
 private int id; // indentifier for this Appointment
 private Date date1; // start date for this Appointment
 private Time time1; // start time for this Appointment
 private Date date2; // end date for this Appointment (may be null)
 private Time time2; // end time for this Appointment (may be null)
 private String text; // the text for this Appointment
 private boolean save; // where this Appointment shoul be deleted (false)
}
```

```
package calendar.models;

public class Note
{
 private int id; // indentifier for this Note
 private Date date; // the date for this Note
 private String title; // the title of this Note
 private String text; // the text for this Note
}
```

```
package calendar.models;

public class Anniversary
{
 private int id; // indentifier for this Anniversary
 private int month; // month for this Anniversary
 private int day; // day for this Anniversary
 private int year1 = 1583; // start year for this Anniversary
 private int year2; // end year for this Anniversary (may be null)
 private String name; // name for this Appointment
}
```

```
package calendar.models;

import java.util.*;
```

```
/**
 * Represents information for a date in the calenndar.
 */
public class MainDate
{
 private Date date;
 private List<Anniversary> anniversaries = new ArrayList();
 private List<Appointment> appointments = new ArrayList();
 private List<Note> notes = new ArrayList();
}

package calendar.models;

/**
 * Represents a calendar for a month.
 */
public class MainModel
{
 private Date date; // the current date
 private MainDate[][] table = new MainDate[6][7];
}
```

11.3.3 DESIGN OF THE DAL LAYER

Data for notes, appointments and anniversaries should be saved persistent, and it is decided to do that by object serialization. Since the program is intended as a personal tool it can be cumbersome to install the program, if you first must create a database that the program has to connect to. If there are relatively few notes and appointments, it is also unnecessary to use a database and it is simpler to serialize the data to a file.

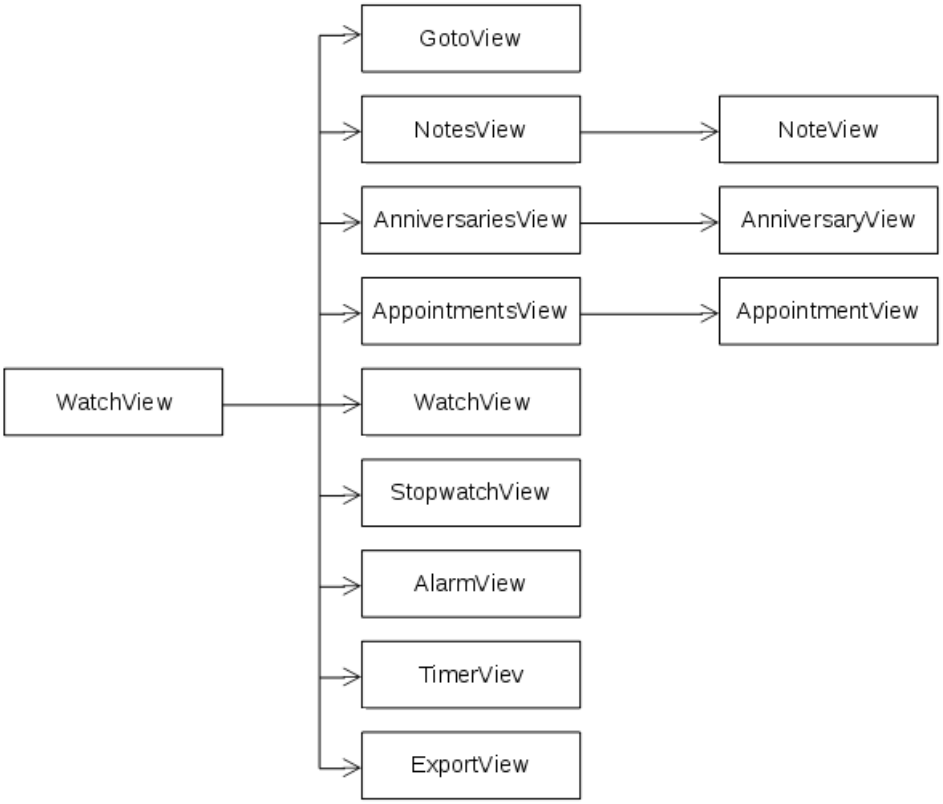
The basis is the following interface, and the methods names should explain what they are used for:

```
package calendar.dal;

import java.util.List;

import calendar.models.*;

public interface Repository
{
 public List<Note> getNotes();
 public List<Note> getNotes(Date date);
 public List<Note> getNotes(Date from, Date to);
 public boolean addNote(Note note);
 public boolean updateNote(Note note);
 public boolean deleteNote(int id);
 public boolean deleteNotes(Date date);
 public boolean deleteNotes(Date from, Date to);
 public List<Anniversary> getAnniversaries();
 public List<Anniversary> getAnniversaries(Date date);
 public boolean addAnniversary(Anniversary anniversary);
 public boolean updateAnniversary(Anniversary anniversary);
 public boolean deleteAnniversary(int id);
 public boolean deleteAnniversaries(Date date);
 public boolean deleteAnniversaries(Date from, Date to);
 public List<Appointment> getAppointments();
 public List<Appointment> getAppointments(Date date);
 public List<Appointment> getAppointments(Date from, Date to);
 public boolean addAppointment(Appointment appointment);
 public boolean updateAppointment(Appointment appointment);
 public boolean deleteAppointment(int id);
 public boolean deleteAppointments(Date date);
 public boolean deleteAppointments(Date from, Date to);
}
```

The program must then instantiate a concrete repository class, what I will call *IORepository*.

11.3.4 DESIGN OF THE USER INTERFACE

The main window is defined by the prototype. In addition, the user interface includes a series of dialog boxes corresponding to the functions of the program, as defined in the analysis. The following diagram shows an overview.



If, for example looking at *NotesView* it is a dialog box that shows a list of all notes within a selected period. *NoteView* is an associated dialog box that is used to maintain a single note.

In addition to the dialog boxes, as shown in the diagram, there will also be a dialog box for maintenance settings. It is still not decided which settings it should be, but the choice to select where to store data, will be one of them.

11.3.5 DESIGN OF THE CONTROL LAYER

Control layer will be thin, and there will be only controller classes the most complex dialog boxes, and for the most part, these are as simple dialog boxes. However, all that is necessary to trigger warnings for appointments and alarms, and thus everything concerning threads are implemented in the controller layer.

11.4 PROGRAMMERING

The programming is carried out in the following steps (iterations):

1. Navigate the calendar and including the implementation of the class *Date*
2. Implentation maintenance of notes and of *IORepository*
3. Implementation of appointments and anniversaries
4. Implementation of all functions relating to the clock and including alarm and timer
5. Implementation of export of a calendar
6. A last iteration and code review

11.4.1 ITERATION 1

The iteration starts with creating a copy of the NetBeans project from the design. The copy is called *Calendar2*.

As the first I have implemented the class *Date*. It is the program's most important and certainly most comprehensive class. Many of the algorithms are complex, and are documented in the code. In addition to the class *Date* the model layer is extended with a class *CalenderException*.

Main class *MainDate* is extended with a constructor, which has as a parameter that is a *Date* object, and the class has a corresponding *get* method.

The class *MainModel* is implemented so that it represents 42 dates, which is the number of dates needed to represent a whole month with 6 rows by 7 columns.

There is created a dialog box where you can select a month and enter a year. Since it is a very simple dialog box there is no controller, but to the package *calendar.views* are added three classes:

1. *GotoEvent* which defines an event for selecting a month
2. *GotoListener* defining a single event handler for an event of the type *GotoEvent*
3. *GotoView* as the dialog box

The class *MainView* from the prototype is modified, which primarily consist of attaching event handlers for the first five buttons in the toolbar.

Then the first iteration in principle is finished, so you can navigate the calendar, but it is important that the result be thoroughly tested, and it will specifically say the class *Date*. The best way to test the result is by using and navigating the calendar:

1. Open the program and validate that the current month is displayed correctly:
   - the right dates
   - the right week days
   - the right week numbers
   - the right selections for Sundays and holidays
2. Navigate a month forward and test similar the result. Repeat it a month at a time for five years.
3. Navigate five years back, or until you reaches the current month. Test the result for each month.
4. Navigate one month back and check the corresponding result. Repeat it a month at a time for five years.
5. Navigate five years forward, or until you have the current month. Check for each year the result.
6. Select the date 1/1 1582 and navigate the calendar forward one month at a time until 1/1 1583. Check that October is correct.
7. Navigate the calendar back a month at a time until 1/1 in 1582 and controls of October is correct.
8. Select the date 1/2 1500 and check that there are 29 days in February.

9. Select the date 1/2 1700 and check that there are 28 days in February.
10. Navigate four years back and make sure that there are 29 days in February.
11. Navigate eight years forward and check that there are 29 days in February.
12. Use the Internet to determine the date of Easter Day for the next 10 years from the current year. Check that the program shows the same dates for Easter Day.
13. Select the date 1/12 9999. Navigate five months back. Navigate the calendar forward as far as possible.
14. Select the date 1/1 0. Navigate five months ahead. Navigate the calendar as far back as possible.

11.4.2 ITERATION 2

The iteration starts with creating a copy of the NetBeans project from the iteration 1. The copy is called *Calendar3*.

I start to implement the class *IORepository*. First I moved the interface *Repository* to the package *calendar.models* and thus it defines which services the DAL layer provides. The interface has been extended with a single method:

```
getNote(int id)
```

which returns a note with a specific id. That must be defined corresponding methods respectively for anniversaries and appointments, but I will first do that in the next iteration.

Next, the class *IORepository* is implemented as follows:

```
public class IORepository implements Repository, Serializable
{
 private static final String filename = "calendar.dat";
 private List<Anniversary> anniversaries;
 private List<Appointment> appointments;
 private List<Note> notes;
 private int lastAnniversary = 0;
 private int lastAppointment = 0;
 private int lastNote = 0;

 public IORepository()
 {
  deSerialize();
 }
```

So far, it is decided that the program's data should be serialized in a file in the user's home directory. The class must implement the interface *Repository*, and to facilitates the work you can take advantage of that NetBeans can automatically generate a stub for all methods defined in the interface. If you right-click on the class name and here choose *Insert Code* you provides an opportunity to choose *Implement Method…* and then NetBeans automatically create a stub for all methods defined in the interface (if you choose them all), and an example is :

```
@Override
public List<Anniversary> getAnniversaries() {
 throw new UnsupportedOperationException("Not supported yet.");
}
```

The methods must then be implemented, and in this iteration, I have only implemented the methods related notes.

As a next step, I have defined the three model classes *Anniversary*, *Appointment* and *Note* (the first two should only be used in the next iteration). All three classes must be defined serializable, and when the individual objects also should be sorted, they must be defined comparable. It is quite simple to implement the three classes, as they mainly consist of *get* and *set* methods and an override of *equals()* and *compareTo()*, but here you can facilitate the work by the same manner as described above to let NetBeans create stubs for each method.

I add a class *DataAdapter* that is used to instantiate an object of the type *Repository*. The class is written as a singleton:

```
package calendar.models;

import calendar.dal.*;

public class DataAdapter
{
 private static DataAdapter instance = null;

 private Repository data;

 private DataAdapter()
 {
  data = new IORepository();
 }

 public static DataAdapter getInstance()
 {
  if (instance == null)
  {
   synchronized (DataAdapter.class)
   {
    if (instance == null) instance = new DataAdapter();
   }
  }
  return instance;
 }

 public Repository getData()
 {
  return data;
 }
}
```

With these classes in place, I can write the code for maintenance of notes. First there is changes in the class *MainView*, so it opens a popup menu if you right-click on a day in the calendar. The menu has three options:

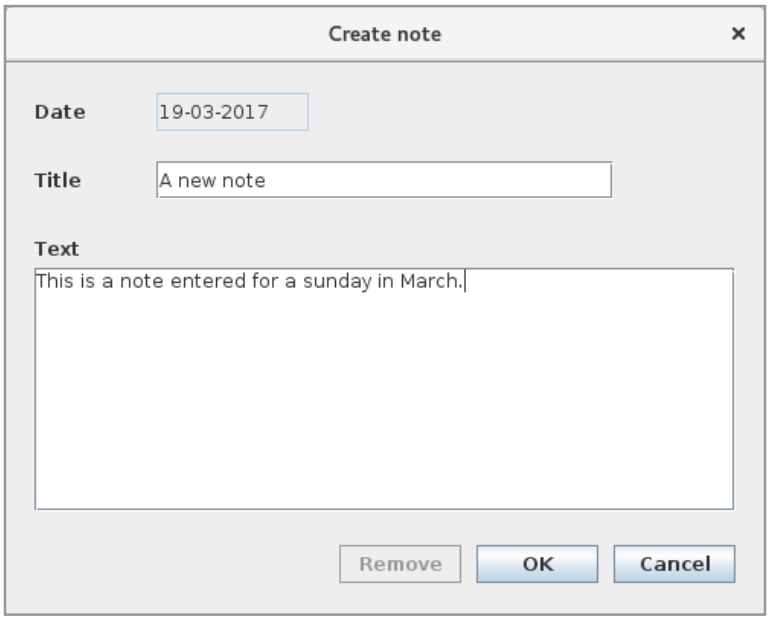
1. Create note
2. Create appointment
3. Create anniversary

A popup menu has the type *JPopupMenu* and the definition of the menu is simple and is done by a call of a method in the class's constructor. For each of the panels that define a day in the calendar is assigned an event handler for the mouse, and the popup menu opens if you right-click a day in the calendar. To maintain notes, I have added the following classes:

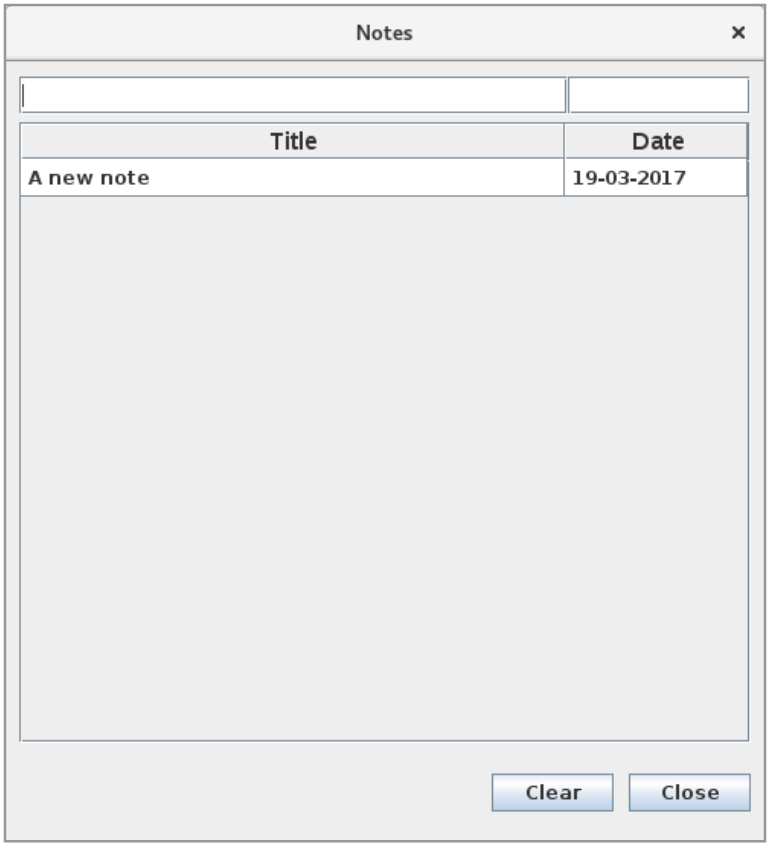
- *NoteController*, which is controller for a dialog box for maintenance of a note.
- *NoteEvent* defining an event that the above controller can fire if adding a note, if a note is changed or if a note is deleted.
- *NoteListener* defining a listener for events of type *NoteEvent*.
- *NoteView* as the dialog box for the controller *NoteController* and is used both to create a note and to edit a note.
- *Filter* that defines a filter for a *JTable.*
- *FilterListener* that defines a listener to a filter.
- *NotesTable*, which defines a data model to a *JTable*.
- *DateRenderer* defining a *CellRenderer* to a *JTable*.
- *NotesView* which opens a dialog box with a *JTable* that shows an overview of all notes.

Finally is added a class *GUI* with helper methods for design of the graphical user interface.

If you right-click a day in the calendar and in the popup menu, select *Create note*, you get a window as shown below (the dialog box *NoteView*):


Create note
Date
19-03-2017
Title
A new note
Text
This is a note entered for a sunday in March.
Remove
OK
Cancel

If you click on the icon for notes in the toolbar, you get the following window (dialog box *NotesView*):



If you double-click on the line for the note, the same dialog as above opens, and you can edit the note and possible. delete it.

The result of the iteration has been tested as follows:

1. Open the program.
2. Right-click on a day the current month. Select *Create note* and create a new note.
3. Navigate the calendar 5 months forward.
4. Right-click on a day the current month. Select *Create note* and create a new note.
5. Navigate the calendar 2 months back.
6. Right-click onn a day the current month. Select *Create note* and create a new note.
7. Navigate the calendar 3 months back.
8. Right-click on the same day as for the first note. Select *Create note* and create a new note.
9. Close the program.
10. Open the program again.

11. Select *Maintenance of notes* in the and test where all notes are shown and in the right order.
12. Double-click on one of the notes and change the text.
13. Double-click on the same note and change the title. Check that the *JTable* is updated.
14. Close the program.
15. Open the program again.
16. Select *Maintenance of notes*.
17. Test the filter.
18. Double-click at the same note as before annd check the changes are preserved.
19. Delete the note.
20. Close the program.
21. Open the program again.
22. Select *Maintenance of notes*.
23. Check that the note is deleted.

11.4.3 ITERATION 3

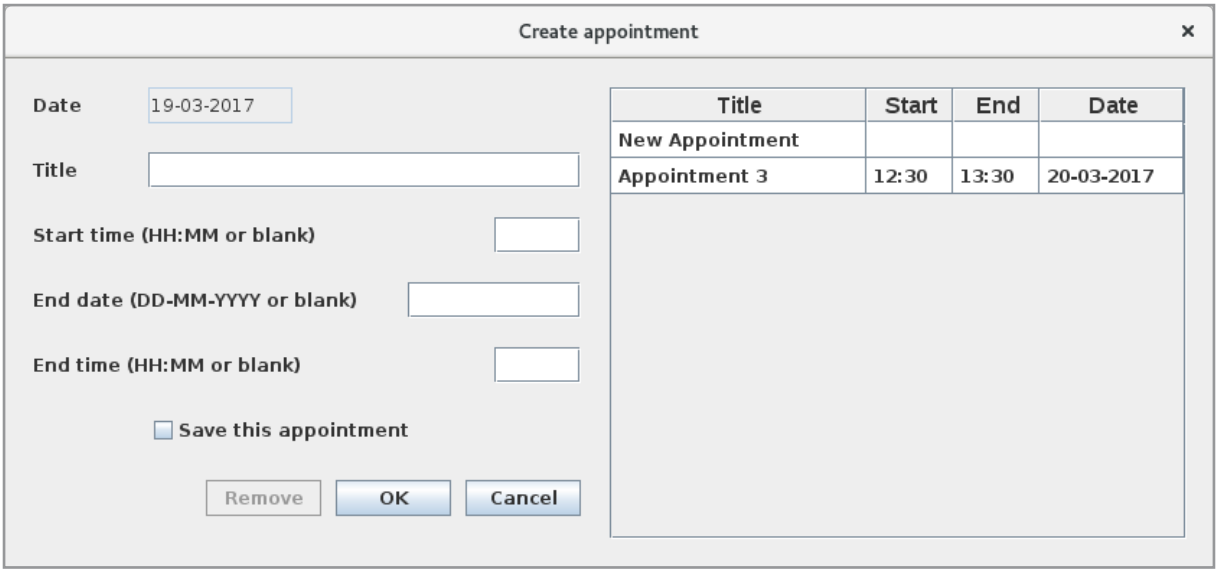
As before the iteration starts with creating a copy of the NetBeans project from the previous iteration. The copy is called *Calendar4*.

In this iteration, the task is to implement the two functions maintenance of appointments and maintenance of anniversaries. In addition, the iteration include that the calendar should displays icons for the days when registered appointments, anniversaries and also notes. Implementing maintenance of appointments is the most comprehensive, so I will start there.

Before that, I have to adjust the model:

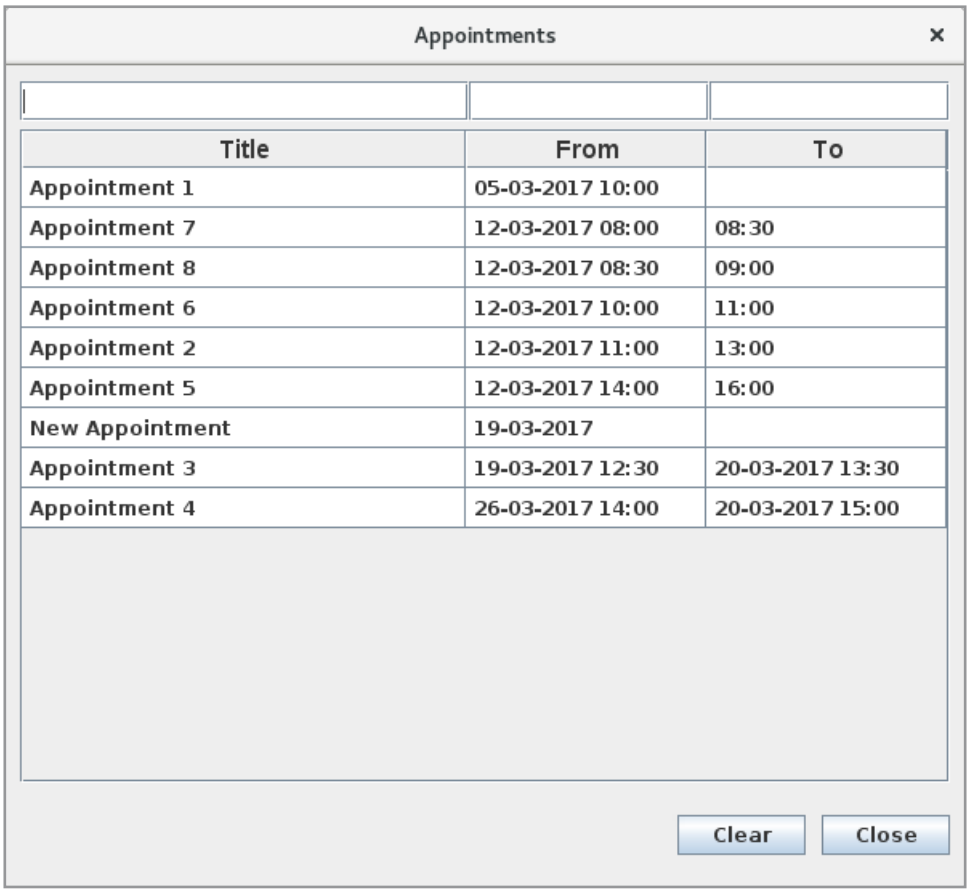
- The class *Time* is implemented, but there has been a change so that a time alone consists of hours and minutes. In this case, the class is used to record times on appintments, and here it makes no sense to register seconds.
- The two classes *Appointment* and *Anniversary* must implement the interface *Comparable*, such that objects can be sorted.
- The interface *Repository* is expanded with two new methods that return either an *Appointment* and an *Anniversary* for a given id.
- The class *IORepository* is updated. First, the two new methods are implemented, and also the code to all other method stubs are written. The code fills, but is basically identical to the corresponding code for notes.

If you right-click a day in the calendar and in the popup menu chooses *Create appointment*, you get the following dialog box:



On the left side you can enter an appointment for the current day. On the right side is a *JTable* showing an overview of appointments for the same day. Here you can edit the individual cells. The purpose is that you can correct times if there are appointments that conflicts.

If you in the toolbar chooses *Maintains of appointments* you get the following window:



which shows an overview of the appointments that are created. Basically it is a *JTable* with a filter. If you double click on a line opens the same dialog as above, so you can modify the appointment and alose delete it.

For the first dialog box is added the following classes and interfaces:

- *AppointmentEvent* defining an event object which is fired when creating a new appointment, when an appointment is changed and when an appointment is deleted.
- *AppointmentListener* which is an interface, which defines the listeners for the events of the type *AppointmentEvent*.
- *AppointmentController* as a controller for the dialog box. It is a relatively extensive class since it must validate where there are conflicting appointments. If you try to save an appointment that conflicts, you get a warning. Two appointments conflicts if they defines overlapping time intervals.
- *AppointmentTable* as a data model for the *JTable* that dialog box contains. The data model must support editing of cells.

- *TimeCellEditor*, which defines a *CellEditor* to edit a time.
- *DateCellEditor*, which defines a *CellEditor* to edit a date.
- *TimeRenderer*, which defines a *CellRenderer* for a column of the type *Time*.
- *AppointmentView* which is the dialog box.

For the second dialog box is added the following classes:

- *AppointmentsTable* that is a data model for dialog's *JTable*.
- *AppointmentsView*, that is the dialog box.

To test the function I have deleted everything that is created by manually delete the file *calendar.dat*. Next, I have completed the following test:

1. Created an appointment for the 8th in the current month from 10:00 to 11:00.
2. Created an appointment for the same day that starts at 12:00 (no end).
3. Created an appointment for the same day from 8:30 to 9:30.
4. Created an appointment for the same day, but without time indication. It results in a conflict, and I have agreed to create the appointment anyway.
5. Closed the program.
6. Open the program again.
7. Open maintenance appointments from the toolbar.
8. Checked that the appointments are sorted correctly.
9. Double-click on the second of the above appointments (that without end time) and entered 14:00 as the end time. When you save you gets a warning about conflicting appointments when the appointment conflicts with the appointment without time indication.
10. Double-click on the appointment without indication of time and set the time interval from 15:00 to 16:00. When saving, there will be no warining – there is no longer conflicting appointments.
11. Created an appointment starting d. 8th and start at 17:00 and ends the next day at 9:00.
12. Opened the list of appointments. Double-click on above appointment and deleted it.
13. Close the program.
14. Open the program again and opened the list with appointments and found that everything looks correct.
15. Created an appointment the 15th, the 22th and the 29th in the same month – all without time indications.

16. Edited the appointment for the 15th with a start and a end time, where the last one should be smaller than the first. It should give an error. Fixed the end time, so it is larger than the start time and saved the appointment.
17. Edited same appointment again and changed the end date to the day before. It should give an error. The click *Cancel* to cancel the change.

Then there is maintenance of anniversaries. This feature is similar to the maintenance of notes, and when then model classes are implemented, the function is implemented by added the following types:

- *AnniversaryEvent* that define an event, when added, modified or deleted an anniversary.
- *AnniverversaryListener* defining listeners for events of the type *AnniversaryEvent*.
- *AnniversaryController* which is controller class for the dialog box *AnniversayView*.
- *AnniversaryView* which is the dialog box for maintenance of anniversaries.
- *AnniversariesTable* defining a data model to a *JTable* with an overview of all anniversaries.
- *IntegerRenderer*, which is used to render cells of the type *Integer*.
- *AnniversariesView*, there is the dialog box, which shows an overview of all anniversaries.

The function is tested by the same pattern as above, and I will not mention the individual test cases here.

The calendar should similar to the prototype display icons for the days when created notes, appointments or anniversaries. If you click on an icon, the application should open the above dialog boxes, respectively for notes, appointments and anniversaries, but they should only show an overview of the items (notes, appointments, anniversaries) for the current day. Furthermore, it is decided that the calendar also must displays an icon for the days that are holidays (the holidays that the class *Date* knows). If you here click on the icon, you should get a simple popup that shows the holy day's name:

In principle, it is easy to implement these changes, but there are changed in many places:

- The interface *Repository* has been extended with three new methods used to test whether a given day have respectively a note, an appointment or an anniversary.
- The class *IORepository* is changed to implement the three new methods.
- The class *MainData* is changed so that it no longer has collections of notes, appointments and anniversaries. Indeed, one can consider completely remove the class when it no longer adds much, but it is provisionally preserved for later changes.
- The classes *NoteView*, *AppointmentView* and *AnniversaryView* are all updated as the default constructor is changed to a constructor with two parameters which are respectively a *Date* and a listener. The aim is that the calendar may be updated if you click on one of the three icons for *Note*, *Appointment* or *Anniversary*.
- The three classes *NotesTable*, *AppointmentsTable* and *AnniversariesTable* is changed where added a default constructor and a constructor that from a *Date* creates a model for that day.
- The three classes *NotesView*, *AppointmentsView* and *AnniversariesView* is changed such that the default constructor now has a listener as a parameter. Again, the reason is that changes in the three collections may be reflected in the calendar (where an icon if necessary must be removed).
- Finally, the class *MainView* is updated to show the icons, and there are changes for the event handling.

I will not show the individual test cases here, but to test the latest addition and thus the entire iteration you needs to create more notes, appointments and anniversaries spread over several days and months thereafter to check that the right icons appear and act as they should. In addition, you must test that the calendar is updated properly if you delete items.

11.4.4 ITERATION 4

The iteration starts with creating a copy of the NetBeans project from the previous iteration. The copy is called *Calendar5*. In this iteration the following functions must be implemented and tested:

1. *Watch*, that show a watch in the window.
2. *Stopwatch*, that implements a stopwatch.
3. *Alarm*, where the user can set an alarm that shows a waring after a given time.
4. *Timer*, where the user can enter a time, and the function shows a warning, when the timer is 0.
5. *Appointments*, where en appointment for the current day must result in a warning, when it's time.

Watch

This is a simple function and should be an on/off function. If you clicks on the icon in the toolbar, the function is turned on, and if you clicks again the function is turned off.

The calendar has a status bar, which shows the current date. It is agreed that the clock should appear immediately afterwards, and the clock will tick every second. It's simple to add the necessary to the *MainView*. To control the clock is added a class *MainController* that has a function that either starts or stops a timer and every time the timer is ticking, is sent an event to the *MainView* (*WatchEvent* and *WatchListener*).

Stopwatch

Then there is the stopwatch. It has been decided that it should only be possible to start one stopwatch. If you click on the button in the toolbar (and the stopwatch not already is started), you get the following window:

where you can start the stopwatch. If you do that, an icon appears in the lower right corner, which indicates that the stopwatch is running and if you click this icon, opens the above dialog boxes again, and you can see what the stopwatch displays and possible stop the clock.

To implement this feature, the model is extended by a class *Stopwatch* representing a stopwatch. There is also added an event object *StopEvent* and an interface *StopListener* (both in the model layer) as the class *Stopwatch* fire an event every time the clock has counted a second forward. It means, when the clock starts it starts a timer that ticks every second. The event is used to update the above dialog box, when the stopwatch ticks. The class *ModelMain* is extended with an object of the type *Stopwatch*.

The above dialog box is called *StopView* and is a simple dialog box without a controller. The dialog box is listening to the *Stopwatch* object.

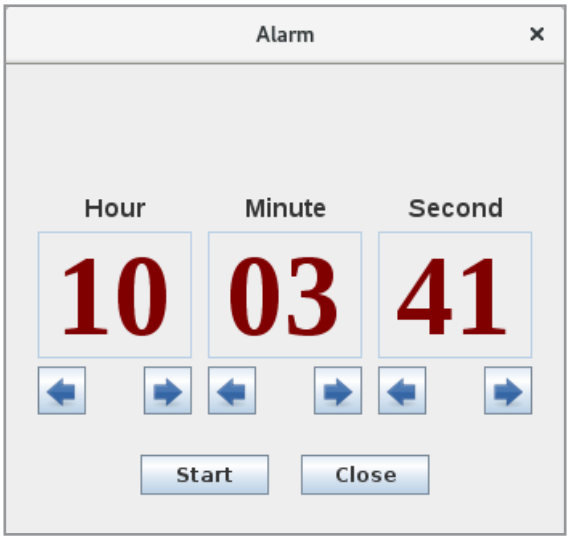
Alarm and Timer

These two features are basically the same function, so I will mention them in the same place.

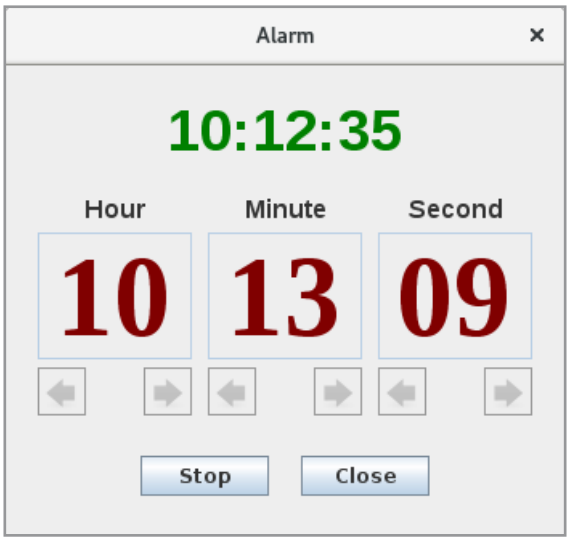
It has been decided that it should only be possible to start one alarm and one timer, and that the two functions should not be persistent. That is, if there is set an alarm or started a timer and the program is closed, the two functions state should not be saved and not restarted if the application is opened again.

The timer is in principle a countdown, and it is decided to call this function a countdown.

If you in the toolbar click the alarm, you get the following window showing the current time. You can then adjust the clock. If you click on start, the window closes and displays an icon in the calendar's lower right corner, indicating that there is set an alarm. When the time of the alarm is triggered, you gets a warning as shown below, and the alarm function is completed.



If you click on the icon in the lower right corner opens the above dialog again:



where one can see the the amount of time to the alarm will occur and also can stop it.

If you choose the countdown you gets the same dialog boxes as above, and the difference is only that the clock start is at 00:00:00 and you then set how long to wait before the function gives a warning. In addition, a second icon in the lower right corner is shown.

To implement the two functions I have added an abstract class *Watch* to the model layer used as a model for both an alarm and an countdown. The difference between an alarm and a countdown is basically how to *reset()* the watch, and that something must be done differently when the clock is started. When this occurs, a timer is started, that ticks every second, and this timer is using a different timer function (*TimerTask* object), depending on whether it is an alarm or a countdown, and the timer function must be implemented in the concrete classes, called respectively *Alarm* and *Counter*. A *Watch* can fire events of type *TimeEvent* which occurs when the clock is started, when the clock is stopped, when the clock is updated, and when there is a timeout (the occurrence of an alarm or countdown is timed out). In the latter case, it also means that the clock must stop. Listeners for these events are defined by the *TimeListener* interface.

The class *MainModel* has objects of the types *Alarm* and *Counter*.

The same dialog box is used for both an alarm and a countdown, and the dialog box is called *WatchView*.

Appointments

The last feature in this iteration relating to appointments, and how to display a warning when an appointment on the current day occurs. Compared to the requirements specification is decided a change for when to display a warning. The reason is that the formulation in the requirements specification can easily lead to too many warnings.

In order to display a warning, the appointment should be defined with a start time. A warning should be displayed when:

1. 1 minute before the appointment starts
2. 5 minutes before the appointment starts
3. 15 minutes before the appointments starts
4. 1 hour before the appointment starts

A warning opens a popup that looks like the above, but there is a *Cancel* button. If you click it, the warning does not appear again.

The logic regarding warnings for appointments is implemented in the class *MainController*. Here the constructor starts a thread running in an infinite loop and every half minute examines whether there are appointments for the current day, for which to display a reminder.

11.4.5 ITERATION 5

The iteration starts as the other iterations and creates a copy of the NetBeans project from the previous iteration. The copy is called *Calendar6*. It is a short iteration with only one new dialog box, where the user must select a period for the calendar to be exported. The calendar is exported in the controller for this dialog box. The iteration only adds to classes to the project: *ExportController* that creates the file, and *ExportView* that is the dialog box.

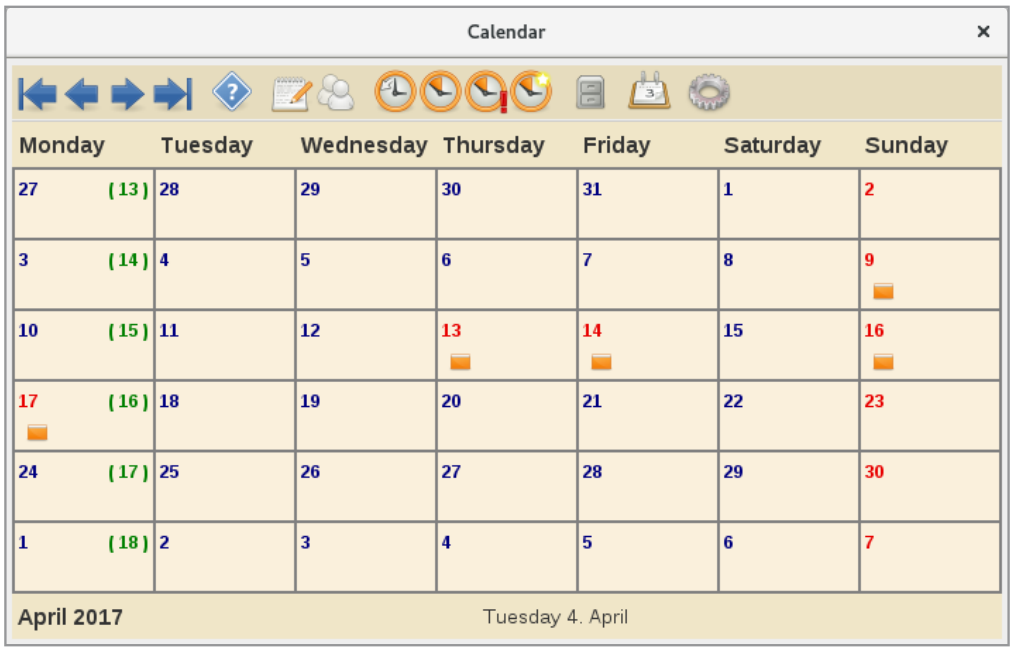
11.4.6 ITERATION 6

Again the iteration starts to make a copy of the NetBeans project from the previous iteration. The copy is called *Calendar7*. After this iteration the programming is finshed, and what is back is the following:

1. A review of the overall look and feel.
2. Implementing of the function options (the last icon in the toolbar).
3. A code review and code documentation.

Look and feel

The goal of it is to get the individual dialogs to look the same, so they use the same colors and fonts, and that the application of colors and fonts are consistent. Fonts and colors are defined as constants in the class *Options*. The work is to go all dialog boxes through and ensure consistent use of fonts and colors. Moreover, the colors are changed, so they are a bit more neutral. Finally, some of the icons are replaced, and the result is a main window as shown below:



Options

If you click on the last icon in the toolbar you will get a popup menu with four menu items:

1. Data location
2. Delete notes
3. Delete appointments
4. Delete anniversaries

If the program is used over a longer period, the data file could be very large, especially if there is stored many notes. There should be an option for deleting old data. The three delete functions work in principle the same way, where you get a dialog box for entering two dates, and so deleting all (notes, appointments or anniversaries) between these two dates. The three dialog boxes are the same and are in principle the same dialog box as *ExportView*. There is therefore written an abstract base class which all four dialogs inherit.

The file *calendar.dat* that contains the program's data is created in the user's home directory. The goal of the first of the above functions is that the user must be able to choose the directory where the file should be saved, and the same user can then use multiple calendars. The path to the file must be stored somewhere, and it has been decided that it should be in a hidden file (*.pa_calendar*) in the user's home directory. The necessary programming to change the directory is placed in the class *DataAdapter*. At the same time two of methods are defined *synchronized* since there is a concurrency problem due to the thread, which tests for appointments.

Code review

After the program is written and finished, I performs a code review. This work can take a long time. Besides updating the comments and review the code the work also to address directly errors and change inconveniences. In this case, I have changed the following.

The class *MainDate* is a wrapper class for a *Date*, and the class is not much more than a very thin encapsulation of a *Date*. Originally the class was intended more functionality, but since it is no longer the case, I have deleted the class. It means:

1. The class has three methods that tests whether a date is attached a note an appointment or an anniversary. These three methods are moved to the class *MainModel* as static methods.
2. All references in the class *MainModel* to *MainDate* are changed to *Date*.
3. All references in the class *MainView* to *MainDate* are changed to *Date*.

Then the class *MainDate* is deleted. It is always dangerous to delete a class at the risk of being overlooked something, but in this case it is simple since the class only is used in *MainModel* and *MainView* and then the compiler wil find all the places where you has to change.

The class *MainView* has a number of very simple inner classes that define event handlers for click on an icon in the toolbar. Most of these classes are removed and replaced by anonymous classes.

The class *MainController* is changed so that the thread that displays messages regarding appointments start with deleting all appointments (those that must be deleted) that are older than the current date. The reason is that if the program has not been used for several days, there may otherwise be appointments that just remains and where time has passed.

11.5 TEST

Now the program is finished and should be tested, and it is in this case a quite extensive work to test the program. The following procedure can be used:

1. The date file is deleted manually.
2. All test cases from all the iterations from the programming are used as test cases for the final test and are performed.
3. The first function in the options menu is used to changing the location of the data file.
4. Create two notes for the current month, an anniversary for the current month, an appointment occurring within 3 minutes and an appointment occurring within ½ hour.
5. Close the program.
6. Open the program again.
7. Test it looks right, especially on the occurrence of warnings for the two appointments.
8. The data file is changed back to the first data file.
9. Examines whether everything looks right.
10. The three delete functions from the options menu are tested.

11.6 DELIVERY

This time it is simple to deliver the program and put it into operation. There should only be used the jar file and an icon. The installation script is similar to scripts as I have shown, and will not be shown here, but the folder *setup* contains the following three files:

1. *Calendar.jar*
2. *calendar.png*
3. *calendar.sh*

where the last one is the installation script. The program can then be installed and put into operation to executes the foolowing command from a terminal:

```
./calendar.sh
```

APPENDIX A

A virtual machine is a program that simulates a physical machine in software and the Java runtime system is such a virtual machine. I will in this appendix provide a brief introduction to what the Java virtual machine is and how it works.

Java's virtual machine is called JVM for *Java Virtual Machine*, and you can think about it as the engine that makes it possible to execute Java programs. It is a program (a machine) that interprets and executes compiled Java programs. Other programming languages such as C and C++ compiles directly to the specific platform, that is to a specific processor's instruction set and to a particular operating system, and the result is called executable code that immediately can be carried out on the concrete machine without the assistance of a virtual machine. This means other things being equal, tthat one gets more effective programs, but there are also significant drawbacks. The main one is lack of portability, where the translated program depends on the machine's processor and operating system, and for the program to run on a different platform, it must be translated again to the new environments.

All this talks of developing a platform independent programming language and it was one of the main ideas that led to the development of Java. The principle is that the Java compiler generates an optimized set of instructions, called bytecode. This code can not be directly executed, since it consists of instructions to a machine that does not exist – instructions for a virtual machine. For a program translated into bytecode to run, there must be running a program on the machine, that simulates a machine whose instruction set is bytecode, and such a program is exactly what JVM is. Platform independence consists in, that any translated Java program can run on any machine that has a JVM running, and the translated bytecode knows nothing about the physical machine's processor or operating system.

As mentioned, the price of this platform independence and program execution by interpretation of a virtual machine is performance. So it was, at least initially, but Java is no longer a purely interpreted language, and most implementations of the JVM is today a mixture of interpretation and execution of compiled code. This means that Java programs today in practice has the same performance as programs whose code is translated into a specific physical platform, also because the runtime system can perform a series of checks and optimization, while the program runs.

It is also worth noting that Java includes an API called JNI, so you can use software modules written in other languages like C and C++. The most important thing with JNI is that you can use the system calls, which possibly not are directly accessible through Java. Finally, it should be mentioned that JNI allows using routines written in C or C++ to situations where the performance requirements are particularly critical. You should be aware that using JNI means sacrificing the platform independence and the program becomes dependent on the platform that the routines are translated to.

In order to achieve this platform independence JVM must be precise and well-defined, which is done with a JVM specification which dictates the format of the bytecode and defines the features and functionality that JVM must have. The specification is public and can be found on Oracle's website, and all can in principle write a JVM.

JVM IMPLEMENTATIONS

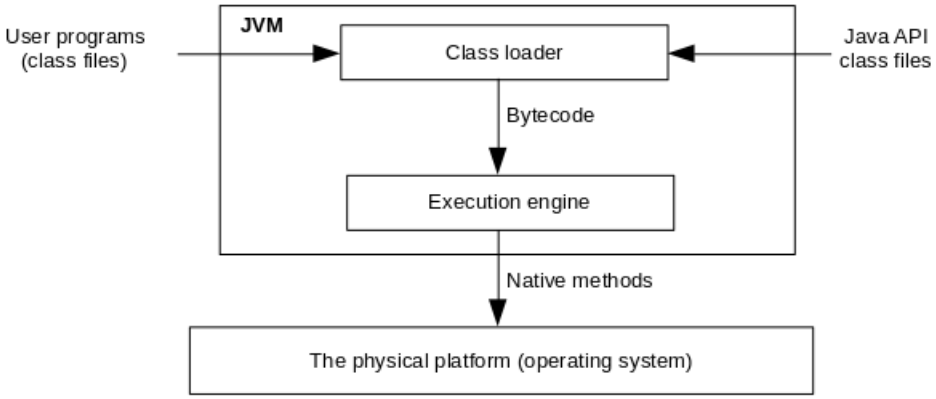
There are several different implementations of JVM, and the most important is probably the JVM from Sun Microsystems, and is often regarded as the reference implementation. After Oracle acquired Sun, it is now their JVM, which is regarded as the reference implementation, but there are certainly others who have implemented the JVM and including IBM, Apple and Hewlett-Packard. However, does the specification of JVM, that Java programs will behave the same regardless of which virtual machine they run on.

In 2006 changed Sun strategy from to control the standard and reference implementation themself to instead to public an open model. This meant several things, including:

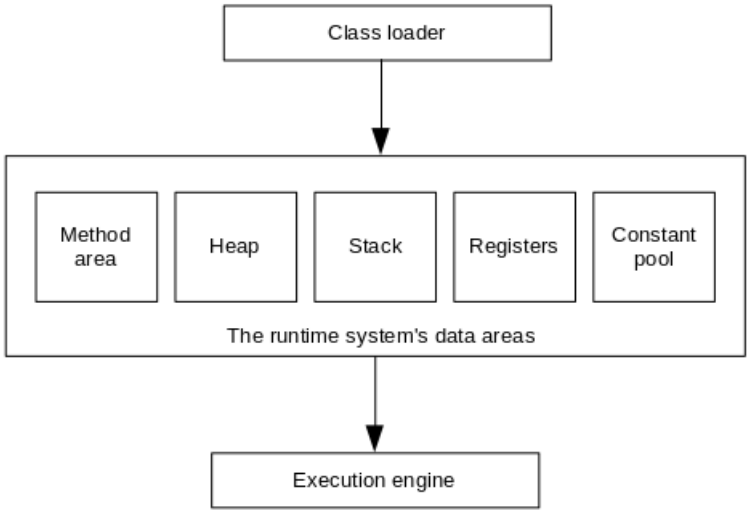
- The full source code was made available to the public, or at least as much of it, as Sun legally could publish because of licenses.
- Changes and extensions of Java has subsequently been handled through the *Java Community Process* (JCP) instead of inside Sun. JCP is an association which is responsible for making decisions about how the language forward should be developed, a development that Oracle has continued and now plays a key role in the decision-making processes.
- Today, is the reference implementation of Java an open source model, called *Open Java Development Kit* and is designated *OpenJDK*.

THE RUNTIME SYSTEM

Every time you perform a Java application the program executes in fact an instance of JVM, and each program has its own JVM instance. A Java program is translated into bytecode, which as stated are instructions for a virtual machine, but these instructions must necessarily somewhere be translated into instructions that the current platform (CPU and operating system) can perform. The translated Java program consists of class files, which mainly contains bytecode, and when the program is executed, these class files are loaded of JVM, which then translates (interprets) the bytecode into instructions for the current machine. JVM can be broadly summarized as follows:



In addition JVM must use memory for the code and temporary data as local variables and so on. The data are stored within the virtual machine's address space:



The heap is an area of available memory and is used to allocate memory at runtime, and it is for objects and arrays. When the program creates an object or an array, it allocates the required memory on the heap, and it is created when the JVM starts. The space that an object or an array uses exist as long as there is a reference to the object, and when it is no longer the case, the space is automatically deallocated by the *garbage collector*. The JVM specification does not tell how the heap will be implemented, and it is also depends on the implementation where the amount of heap is fixed or may change accordance with the demand, but is there not enough heap for a program, you get an *OutOfMemoryError* exception.

The stack is used to maintain information on the methods being performed, and every time you call a method the runtime system creates an activation block on the stack, that contains local variables, parameters and return value and the operand stack used by the runtime system. An activation block is created when a method is called and removed again when the method terminates for one reason or another. Each thread has its own stack, and for each stack there is only one activation block which is active at a time, and it is the block for the method the thread is currently performing. Just as for the heap tells the specification nothing about how the stack must be implemented, and it can be both of fixed size or expand dynamically, but requires a thread more stack than is available, the program will stop with a *StackOverflowError* exception.

The method area is shared by all threads and are used for information on methods and their bytecode, data fields and constructors.

JVM has registers in the same way as a physical machine. They reflect the virtual machine's current state and are constantly updated, as the code is executed. The main register is perhaps the program counter which contains the address of the next JVM instruction to be performed. Other data is a pointer to the method currently performed, a pointer to the first local variable for the current method, and a pointer to the top of the operand stack.

The last are constant pool and is used for constants.

Then there is the garbage collector. As mentioned allocates Java automatic memory to an object when it is created with *new*, and the allocated memory is again deallocated automatically when there are no longer references to the object. Everything is handled by the garbage collector, which is a program that constantly runs and will manages the heap. This is done using a table of pointers that point to the individual objects on the heap. We call these pointers for soft pointers, because instead of pointing directly at the concrete objects they points to the object's references. The garbage collector runs in the background in its own thread and performs periodically checking the object's state, and when there are no longer references to an object, the space which the object has used on the heap is released, and the pointer is removed from the table. When the runtime system continuously allocates objects and deallocates them again, the heap may be fragmented, and if it happens the garbage collector defragments the heap (which means that the objects are move) and thus gather all the available space on the heap in a large contiguous area. It is from there that garbage the collector takes its name.