Laboratory 3

Threads of execution in Java SE – Classic synchronization mechanisms

1. Laboratory objectives

- The following concepts
    - classic synchronization mechanisms;
    - groups of threads;
    - swing and threads of execution;
    - communication between the threads of execution.

2. Theoretical consideration and examples of applications. Classic synchronization mechanisms between threads of execution

2.1 Mutual exclusion using the synchronized keyword

One of the most important issues to consider when working with threads of execution is how they access common resources. Thus, when two or more threads share a common resource, they must synchronize their access to that resource.

An example of such a situation is that of the consumer / producer threads. One thread enters data into a *buffer*, and the second reads data inputted by the first thread into the *buffer*. Obviously, the two threads do not have to access the *buffer* simultaneously and must have synchronized access to it.

Collision prevention in *Java* is achieved by using synchronized methods. A method is synchronized when it faces the ***synchronized*** keyword.

Example:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Only one thread can access a synchronize method of an object at a time. Each object that has a synchronized method includes (automatically) a monitor or a lock. When a synchronized method of the object is called, the monitor is acquired by the thread that accessed the method. As long as the synchronized method is running, no other synchronized method of the respective object can be called by another thread.

Observation:

When a thread tries to access a synchronized method of an object that has the monitor acquired by another thread, then it is stuck waiting for the monitor to be released.

Example 1:

Specifications: It is required to implement an application (Java 2 SE) where the main thread starts two other threads responsible for writing and reading a group of text messages from a text file. The access to the file will be synchronized - because it is desired to read the whole message group - by using synchronized methods implemented in a class that is responsible for the actual access to the file. The "writer" thread will save a new line in the file, the current date and time and on another line, a random integer within the interval [0, 100], every 2 seconds. The "reader" thread will read the last message written in the file (random number) and will display it in the console next to the date and time when the reading was performed, every 3 seconds.

To **design** the application, the class diagram and the sequence diagram will be used. The class diagram is shown in Figure 3.1.

Required: create the sequence diagram.

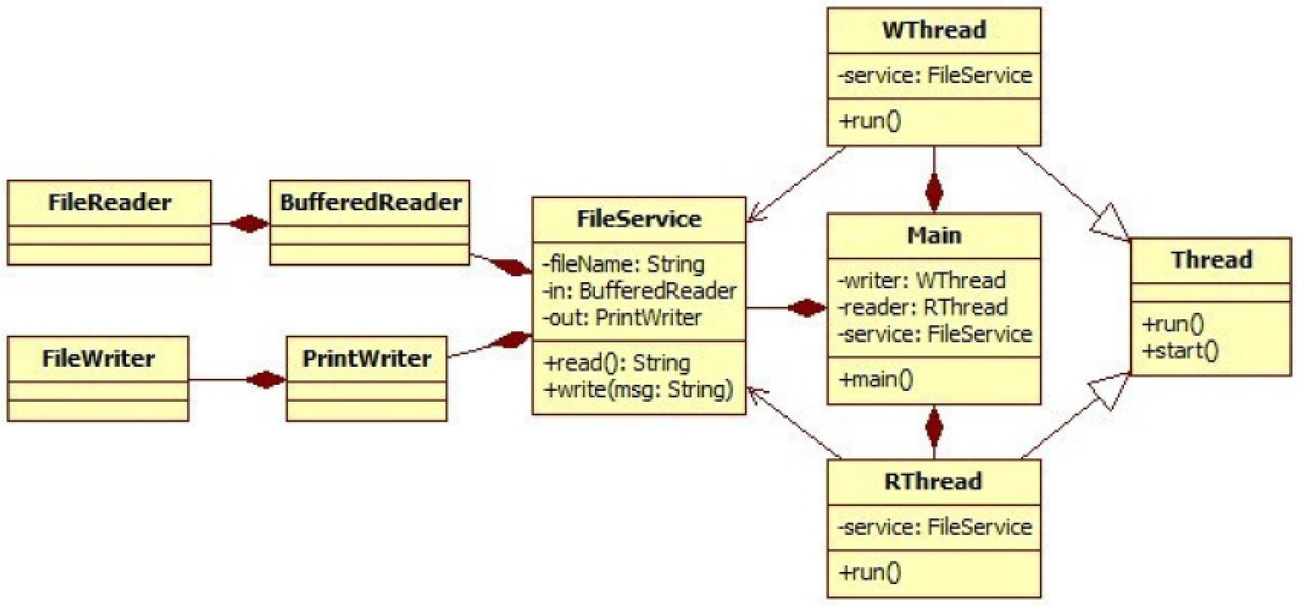

Figure 3.1 The class diagram

Application **implementation** is presented in the code sequences 1, 2, and 3.

Code sequence 1 : Main class

```
public class Main {
      private static boolean stopThreads = false;
      public static void main(String[] args){
            FileService service = new FileService("messages.txt");
            RThread reader = new RThread(service);
            WThread writer = new WThread(service);
            reader.start();
            writer.start();
      }
      public static boolean isStopThreads(){
            return stopThreads;
      }
}
```

Code sequence 2: FileService class

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
public class FileService {
      String fileName;
      BufferedReader in;
      PrintWriter out;
      public FileService(String fname){
            this.fileName = fname;
            try {
                out = new PrintWriter(new FileWriter(fileName, true));
                in = new BufferedReader(new FileReader(fileName));
            } catch (Exception e) { e.printStackTrace();}
      }
      public synchronized void write(String msg){
            Date date = new Date(System.currentTimeMillis());
            out.println("Date:  " + date);
            out.println("Message:  " + msg);
            out.flush();
      }
      public synchronized String read() throws IOException{
            String iterator, last="no message to read";
            while((iterator = in.readLine()) != null){
                  last= new Date(System.currentTimeMillis()) + " - "
                  + iterator;
            }
            return last;
      }
}
```

Code sequence 3: WThread and Rthread classes

```
public class WThread extends Thread{
      FileService service;
      public WThread(FileService service) {
            this.service = service;
      }
      public void run(){
            while(!Main.isStopThreads()){
                  String msg=
                        String.valueOf(Math.round(Math.random()*100));
                  service.write(msg);
                  try {
                        Thread.sleep(2000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}

public class RThread extends Thread{
      FileService service;
      public RThread(FileService service) {
            this.service = service;
      }
      public void run(){

            while (!Main.isStopThreads()){
                  try {
                        String readMsg = service.read();
                        System.out.println(readMsg);
                        Thread.sleep(3000);
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

2.2 Synchronizing of threads using *wait*(), *notify*() and *notifyAll*()

The *wait*() and *notify*() methods are used to lock and unlock threads of execution. An important observation regarding these two methods is that they belong to the *Object* class. These two methods manipulate object monitors and in turn, monitors are at the level of any object.

The *wait*() and *notify*() methods can only be called from within blocks or synchronized methods. These methods, for a given object, can only be called by the holder of the object monitor. When a thread is blocked by calling the *wait*() method, the monitor that is held by the respective thread is released.

Observation:

The *wait*(*millis*) method can be used instead of the *Thread.sleep*(*millis*) method, with the advantage that the thread can be unlocked before the waiting time has expired, by calling the *notify*() method, which in the case of the *sleep*() method is not possible.

Example 2:

Specifications: To design and implement a "*producer-consumer*" type application. The application will have three threads, in addition to the main thread: one *producer* thread and two *consumer* threads. The *producer* thread will add a *Double* object to a list, as a task for consumers, and then notify the other two threads about its existence. The *consumer* threads will wait for the notification, after the object will be extracted from the list and will display its numerical value on the screen. The application will be implemented through the ***wait/notify*** mechanism.

Design: The class diagram and the state machine diagram will be used.

Required: drawing the two diagrams.

Implementation: see Code Sequences 4, 5 and 6.

Code sequence 4: Main class

```
public class Main {
      public static void main(String[] args){
            Buffer b = new Buffer();
            Producer pro = new Producer(b);
            Consumer c = new Consumer(b);
            Consumer c2 = new Consumer(b);
            pro.start();
            c.start(); c2.start();
      }
}
```

Code sequence 5: Buffer Class

```
class Buffer{
      ArrayList<Double> content = new ArrayList<Double>();
      synchronized void put(double d){
            content.add(new Double(d));
            notify();
      }
      synchronized double get(){
            double d=-1;
            try{
                  if(content.size()==0) wait();
                  d = (content.get(0)).doubleValue();
                  content.remove(0);
            }catch(Exception e){e.printStackTrace();}
            return d;
      }
}
```

Code sequence 6: Producer and Consumer classes

```java
class Producer implements Runnable{
	private Buffer bf;
	private Thread thread;
	Producer(Buffer bf){this.bf=bf;}
	public void start(){
		if (thread==null){
			thread = new Thread(this);
			thread.start();
		}
	}
	public void run(){
		while (true){
			bf.put(Math.random());
			System.out.println("Producer "+thread.getName()+
				" gave a task.");
			try{
				Thread.sleep(1000);
			}catch(Exception e){e.printStackTrace();}
		}
	}
}
//-----------------------------------------------------------------
class Consumer extends Thread{
	private Buffer bf;
	Consumer(Buffer bf){this.bf=bf;}
	public void run() {
		while (true){
			System.out.println("Consumer "+this.getName()+
				" received >> "+bf.get());
		}
	}
}
```

2.3 Synchronization between threads of execution using the join() method

The *join*() method belongs to the *Thread* class and is used to make a thread of execution to wait for another thread to finish.

Example 3:

The application from Code Sequence 7 is given.

Specifications: The application will be modified so that the thread "Thread 1" will determine all the dividers of a natural number greater than 50,000, and the thread "Thread 2" those of a number greater than 20,000. Before completing its execution, "Thread 1" will assign the sum of the dividers determined to a static variable of the main class, initially set to 0. "Thread 2" will determine the sum of its own divisors and add it to the static variable mentioned above, but not before it is set by "Thread 1". The result will be displayed in the console.

Required: Modified source code of the application.

Code sequence 7: source code for *join*() method testing

```
class JoinTestThread extends Thread{
	Thread t;
	JoinTestThread(String n, Thread t){
		this.setName(n);
		this.t=t;
	}
	public void run() {
		System.out.println("Thread "+n+" has entered the run() method");
		try {
			if (t != null) t.join();
			System.out.println("Thread "+n+" executing operation.");
			Thread.sleep(3000);
			System.out.println("Thread "+n+" has terminated operation.");
		}
		catch(Exception e){e.printStackTrace();}
	}
}
public class Main {
	public static void main(String[] args){
		JoinTestThread w1 = new JoinTestThread("Thread 1",null);
		JoinTestThread w2 = new JoinTestThread("Thread 2",w1);
		w1.start();
		w2.start();
	}
}
```

2.4 Groups of threads of execution

Each thread belongs to a group (*threadgroup*). This may be the *default* group or may belong to an explicitly specified group at the time it is created. A once created thread cannot pass from one group to another. If not specified, the default group that a thread belongs is the system group. Also in turn, the newly created groups are hierarchically grouped and at the top are the same system group.

Working with groups of threads is useful when you want to control all threads belonging to a group by a single command (method call).

Example 4:

The example from Code Sequence 8 is proposed.

Required: Test the application. Develop the specifications of the application. Make the class diagram and the sequence diagram.

Code sequence 8: example for working with groups of threads of execution (ThreadGroup)

```
class ThreadEx extends Thread {
	boolean stop;

	ThreadEx(ThreadGroup tg, String name) {
		super(tg, name);
		stop = false;
	}

	public void run() {
		System.out.println(Thread.currentThread().getName() + " ON.");
```

```
			try {
				for (int i = 1; i < 1000; i++) {
					System.out.print(".");
					Thread.sleep(250);
					synchronized (this) {
						if (stop)
							break;
					}
				}
			} catch (Exception exc) {
				System.out.println(Thread.currentThread().getName() + " intrerupt.");
			}
			System.out.println(Thread.currentThread().getName() + " The end.");
		}

		public void stopThread() {
			stop = true;
		}
}
public class ThreadGroupDemo {
		public static void main(String args[]) throws Exception {
			ThreadGroup tg = new ThreadGroup("Group of threads");
			ThreadEx fir1 = new ThreadEx(tg, "ThreadEx #1");
			ThreadEx fir2 = new ThreadEx(tg, "ThreadEx #2");
			ThreadEx fir3 = new ThreadEx(tg, "ThreadEx #3");
			fir1.start();
			fir2.start();
			fir3.start();
			Thread.sleep(1000);
			System.out.println(tg.activeCount() + " Thread in group.");
			Thread thrds[] = new Thread[tg.activeCount()];
			tg.enumerate(thrds);
			for (Thread t : thrds) {
				System.out.println(t.getName());
			}
			fir1.stopThread();
			Thread.sleep(1000);
			System.out.println(tg.activeCount() + " Thread in group.");
			tg.interrupt();
		}
}
```

2.5 Swing and threads of execution

The methods of the *swing* classes are not *thread-safe*, which means they are not designed if they are called simultaneously by several threads. So once the thread started for handling Swing events (calling *setVisible*(), *pack*() or any other method that makes a window visible), no method of securely *swing* classes can be called anymore. For modifying a component by a thread, the *invokeLater*() method is used as follows:

```
SwingUtilities.invokeLater(new Runnable() {
	public void run() {
		some_window.repaint();
	}
});
```

2.6 Communication between threads

The communication between the threads can be achieved by the help of the *pipe* streams. *Java* language allows the construction of *byte*-oriented *pipe*s using the *PipedInputStream*/*PipedOutputStream* classes and character-oriented streams using the *PipedReader*/*PipedWriter* classes.

Example 5:

Specifications: To design and implement a *Java 2 SE* application where the main thread creates two other threads that communicate through *pipes*. One of the two threads will write in the *pipe* a message consisting of a random number in the interval [0,10], every 4 ms. The other thread will read the messages and display them in the console.

Design: It is required to make the class diagram and the communication diagram.

Implementation: The source code of the application is given in Code Sequence 9.

Code sequence 9: example for communication between threads of execution

```
import java.io.PipedOutputStream;
import java.io.*;
public class Main{
      public static void main(String args[]) {
            ReadThread rt = new ReadThread();
            WriteThread wt = new WriteThread();
            try{
                  rt.conect(wt.getPipe());
                  rt.start();wt.start();
            }catch(Exception e){e.printStackTrace();}
      }
}

class WriteThread extends Thread{
      private PipedOutputStream po;
      WriteThread(){
            po = new PipedOutputStream();
      }
      public void run(){
            try{
                  while (true){
                        int d = (int)(10*Math.random());
                        System.out.println("Writing Thread is sent : "+d);
                        po.write(d);
                        sleep(400);
                  }
            }
            catch(Exception e){
                  e.printStackTrace();
            }
      }
      PipedOutputStream getPipe(){return po;}
}

class ReadThread extends Thread{
      private PipedInputStream pi;
      ReadThread(){
            pi = new PipedInputStream();
      }
      public void run(){
            try{
                  while (true){
                        if (pi.available()>0)
                        {System.out.println("Read Thread is received :
                              "+pi.read());}
                  }
            }catch(Exception e){}
      }
      void conect(PipedOutputStream os)throws Exception{
      pi.connect(os);
      }
}
```

3. Developments and Tests

3.1 Application 1:

3.3.1 Problem Statement: Modify the application in the first example so that synchronized blocks (in the body of *run*() methods) are used instead of synchronized methods.

3.1.2 Required:

a) class diagrams;

b) state machine diagrams;

c) application source code.

3.1.3 will test the following:

a) writing the messages correctly in the text file;

b) the correct reading of the last message in the text file;

c) the correct synchronization between the two threads of execution.

3.2 Application 2:

3.2.1 Problem statement (specifications): A *Jave 2 SE* application with a graphical interface similar to the one in Figure 3.2 will be implemented. Each square in the graphical interface will have a thread of execution (via the *Observer*/*Observable* mechanism), which will move its own square to the exit of the opposite enclosure. Moving the squares will only start after the "s" key is pressed. The speed of the squares will be set randomly when the application is started. Square 3 will have a maximum speed that is equal to square 1 and will not be allowed to pass.

The three squares are not allowed to touch each other, which is why it is necessary to synchronize the access through the narrow area between the two enclosures. Thus, the first square that reaches the dotted line will block the square from the opposite enclosure to the dotted line from that side (to be able to avoid it). This synchronization will be implemented through ***synchronized*** methods or blocks.

If the first square passing through the narrow area is square 1, it will not continue towards the exit but will wait for the arrival of square 3, after the dotted line. Only squares 1 and 2 must leave the enclosure. This synchronization will be implemented using the ***join***() method.

3.2.2 Required:

a) class diagrams;

b) sequence diagrams;

c) state machine diagrams;

d) application source code;

3.2.2 Will test the following:

a) the correct creation of the graphical interface (before pressing the "s" key);

b) the correct mechanism implementation for moving the squares from one enclosure to another (Observer / Observable);

c) the correct synchronization when passing through the narrow area;

d) the correct evasion of the squares of the opposite enclosure;

e) final synchronization (waiting for square 3).

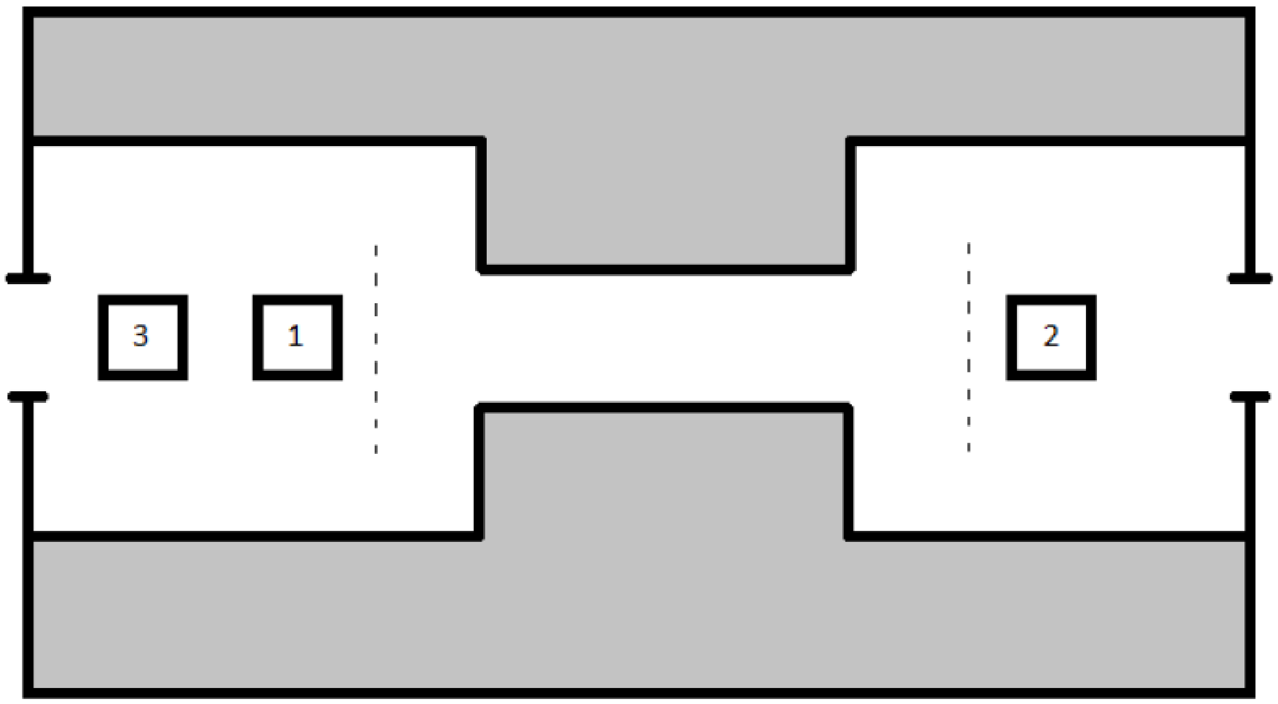

Figure 3.2 Application 2 interface

4. Knowledge verification

1. How can you use the *synchronized* keyword?

2. What is the *join*() method used for?

3. Under what conditions can the *wait*() and *notify*() methods be used?

4. How is the communication between threads achieved?