Laboratory 2

Threads of execution in Java SE – General Notations

1. Laboratory objectives

- learning the concepts
  - threads of execution
  - the states of the threads
- threads management
  - creating threads
  - starting threads
  - stopping threads
- setting the priorities of threads

2. Theoretical consideration and examples of applications

Threads of execution are sequences of a program (process) that are effectively or virtually executed in parallel within a single process. These threads are created and then started, being allowed to evolve freely or controlled.

2.1 The states of a thread

A thread of execution can be in one of the following states:

1. Non-existent - the thread object was not created.
2. New - the thread object was created but not started yet.
3. Runnable - the thread is in the state in which it can be run when the processor becomes available.
4. Execution=Running - the thread is in execution.
5. Dead - the normal path for a thread to end is by exiting the *run*() method.
6. Waiting=blocked (waiting = blocked) - the thread is blocked and cannot be run, even if the processor is available, until a condition for unlocking it is met.

A thread can enter the blocked state the following situations:

- the thread called the *Thread.sleep*(*sec*) method which causes the thread to block for a specified number of seconds;
- the thread has called a synchronized method that has the monitor captured by another thread;
- the *wait*() method was called;
- the thread waits after an operation (with flows) input-output [ORA, 2012b];

2.2 Creation and start of execution threads

When starting a *Java* application (program) start with the execution of the *main*() method. To create additional threads in *Java*, you can use two methods: extending the *Thread* class or implementing the *Runnable* interface. The two mechanisms responsible for creating threads in *Java Standard* will be described.

2.2.1 Extending Thread class

A thread can be built from a user-defined class and inheriting (extending) the *Thread* class. The Thread class is defined within the *java.lang* package (one of the *Java Standard* packages) and this class defines all the properties and functionalities of a thread. A class that extends the *Thread* class becomes a thread type class so that an instance object of this class can be used to launch a thread.

Example:

```
class Sorting extends Thread {
      public void run(){
            // activities carried out by the thread of execution
            …
      }
}
```

A thread type class will have to override the *run*() method, inherited from the *Thread* class. Within this method the sequence of instructions that will be executed by the thread of execution will have to be defined. To launch a thread, use the *start*() method, also inherited from the *Thread* class. This method is responsible for initializing all the mechanisms necessary to execute a thread, and then it automatically calls the *run*() method.

Example:

Specifications: It is required to design an application that creates, besides the main (main) thread, another 3 threads. Each thread implements a counter that it periodically increments by counting the iterations. The thread writes on its screen its name and index of the interaction. Threads will be built by extending the *Thread* class.

Proposing or the **design** phase of the application: the class diagram, presented in Figure 2.1) and the activity diagram, presented in Figure 2.2.

Proposing Code sequence 1 for the **implementation** phase of the application. In code sequence 1, if the call of the *start*() methods is replaced with that of the *run*() methods, the execution of the three instruction sequences are on a single thread of execution, i.e. on the main thread. The latter is created by default when executing the *main*() method.

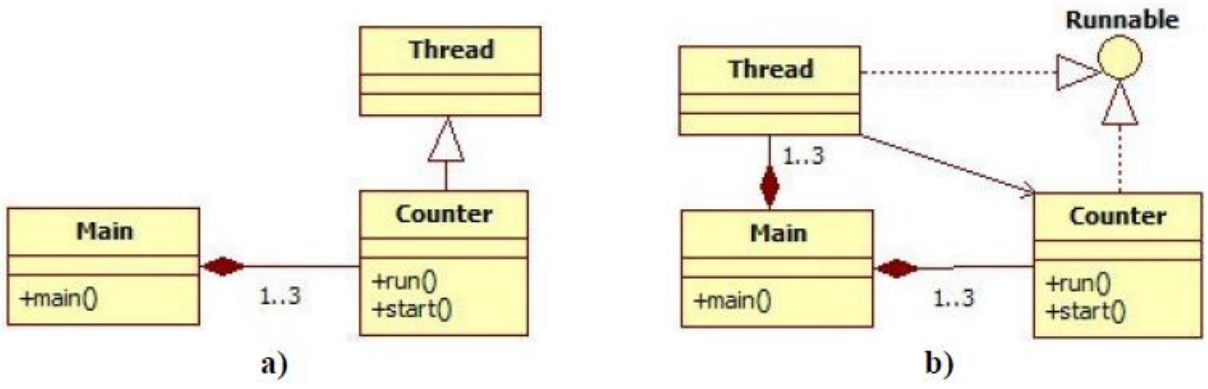

Figure 2.1 Class diagrams

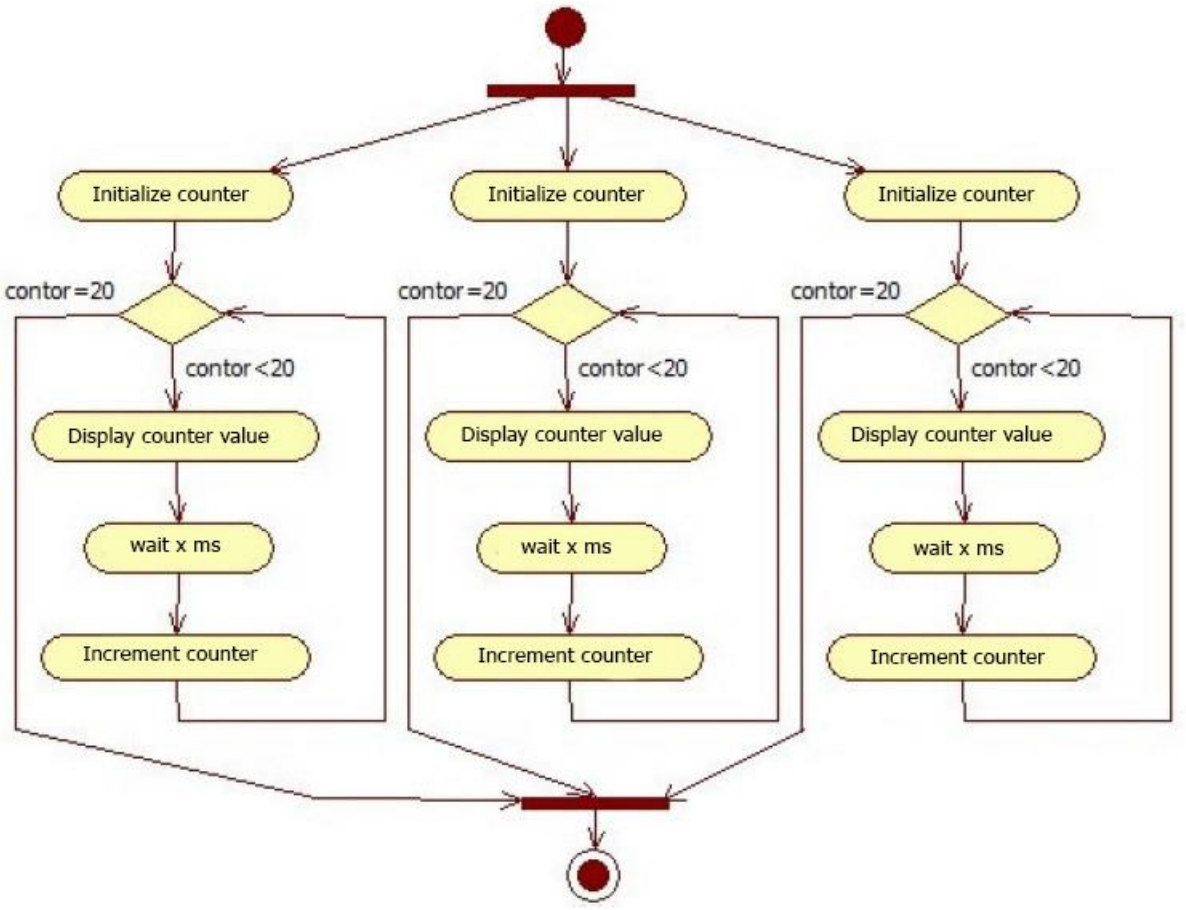

Figure 2.2 Activity diagram

Code sequence 1

```
class Main{
          public static void main(String[] args){
                        Counter c1 = new Counter("counter1");
                        Counter c2 = new Counter("counter2");
                        Counter c3 = new Counter("counter3");
                        c1.start(); //c1.run();
                        c2.start(); //c2.run();
                        c3.start(); //c3.run();
              }
}
class Counter extends Thread {
            Counter (String name){
                        super(name);
             }
            public void run(){
                        for (int i =0; i<20; i++){
                                    System.out.println(getName() + " i = "+i);
                                    try {
                                                Thread.sleep((int)(Math.random() *
1000));
                                     }catch (InterruptedException e)
{e.printStackTrace();}
                        }
                        System.out.println(getName() + " job finalised.");
                }
}
```

2.2.2 Implementing Runnable interface

Since *Java* does not support multiple inheritance, the implementation of the *Runnable* interface is used to create threads of execution that can inherit another class at the same time.

Example:

Specifications: It is required to design an application that creates, besides the main (main) thread, another 3 threads. Each thread implements a counter that periodically increments by counting the iterations. The thread of execution writes on its screen its name and index of the interaction. The threads of execution will be built using the *Runnable* interface.

The **design** of the application was carried out by the class diagrams in Figure 2.1.

The activity diagram is the same as in the previous application (see Figure 2.2).

The **implementation** of the application is presented in Code Sequence 2.

Code Sequence 2

```
public class CounterRunnable implements Runnable {
          public void run(){
                      Thread t = Thread.currentThread();
                   for(int i =0; i<20; i++){
                               System.out.println(t.getName() + " i = "+i);
                            try { Thread.sleep((int)(Math.random() * 1000));}
                            catch (InterruptedException e) {e.printStackTrace();}
                       }
              System.out.println(t.getName() + " job finalised.");
            }
}
class Main{
          public static void main(String[] args) {
                       CounterRunnable c1 = new CounterRunnable();
                       CounterRunnable c2 = new CounterRunnable();
                       CounterRunnable c3 = new CounterRunnable();
                       Thread t1 = new Thread(c1,"conuter1");
                       Thread t2 = new Thread(c2,"conuter2");
                       Thread t3 = new Thread(c3,"conuter3");
                       t1.start();
                       t2.start();
                       t3.start();
              }
}
```

When calling the *start*() method within the *Thread* object, the *run*() method within the transmitted object will be executed as an argument when creating it.

Alternatively, you can create a *start*() method in the *CounterRunnable* class (see thr example below), where the thread will be created and started. Thus, the threads of execution implemented by extending the *Thread* class to those created by implementing the *Runnable* interface can be treated individually.

Example:

```
public void start(){
        new Thread(this).start();
}
```

2.3 Stopping threads of execution

The recommended way to terminate a thread is through normal return, that is, executing the *return* statement from *run*() or terminating all the thread instructions. Within the *run*() method, conditional blocks can be added to determine ending the thread of execution, respectively exit from the *run*() method.

When a thread is blocked due to the execution of *wait*(), *sleep*(), *join*() or I / O operations, it cannot verify any conditions. In order to force the termination of a thread that is blocked, the *interrupt*() method can be used, *but this is not the most appropriate method*. This method causes throwing an *InterruptedException* that has to be treated.

A recommended method of terminating a thread of execution is to construct a *stop*() method, which will use a logical variable to control the execution.

Example:

```
public void stop(){
        this.loop=false;// loop used as a logic variable for loopback
}
public void run(){
        while (this.loop){
        //... sequence of instructions
        }
}
```

2.4 Setting the thread of execution priorities

The priority of a thread of execution provides the scheduler with information about the importance of the respective thread. The default scheduler sorts the *ready-to-execute* list according to the priorities set. The *setPriority*() method is used to set the priority of a thread, and to get the priority of a thread the *getPriority*() method is used, the methods are part of the *Thread* class. Threads can receive priorities between *MIN_PRIORITY*=1 and *MAX_PRIORITY*=10, these are constants defined in the *Thread* class. The default priority of a thread is the priority of the thread that created it.

The priorities used by the *Java* virtual machine depend on the facilities offered by the operating system where it is implemented.

3. Developments and tests

3.1 Application 1:

3.1.1 Requirements: To implement an application for determining the number of logical processors (cores and / or threads) of the computer system.

3.1.2 Specifications:

1. The application will be implemented in *Java 2 SE*.
2. The application will start multiple threads, each with a different priority. The threads of execution will increment one counter.
3. The application will have a graphical user interface where the values of the counters will be displayed as progress bars.
4. The number of "faster" bars, which will advance relatively suddenly - despite different priorities - will give the number of logical processors.

3.1.3 Application design:

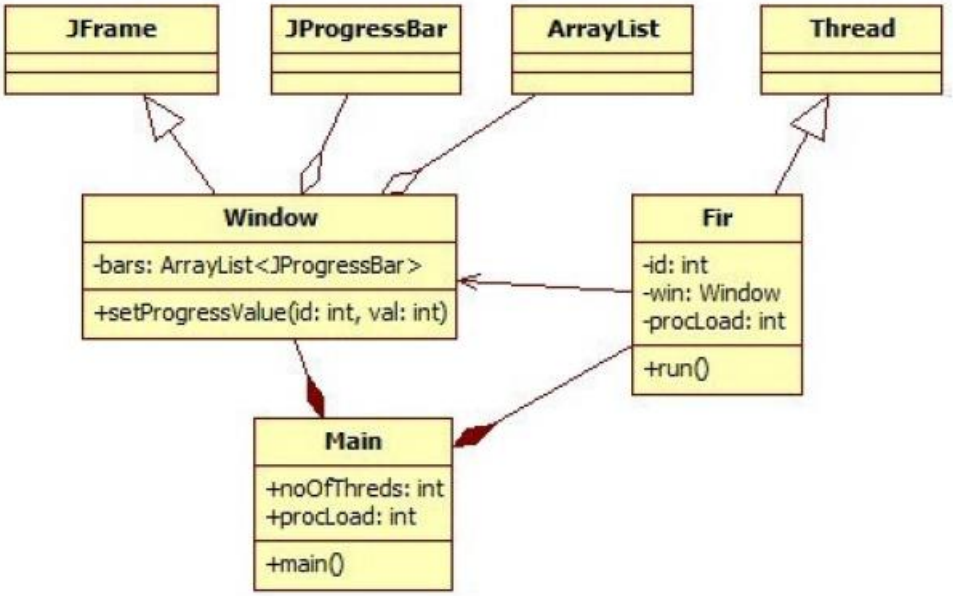

Figure 3.1 Class diagrams

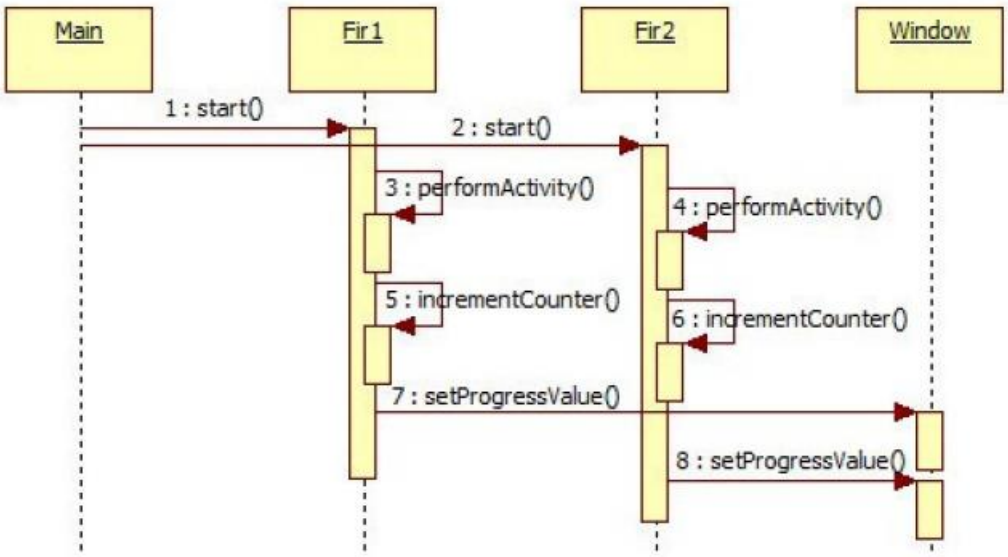

Figure 3.2 Sequence Diagram

In the diagram in Figure 3.1 we considered two threads created by the application. A thread of execution will work iteratively (in the loop) until the associated progress bar is filled. On each iteration, thread of execution will execute an activity (for loading the processor), then increments the counter, respectively, updates the value in the associated progress bar by the *id* parameter.

3.1.4 Implementation:

Code sequence 3: Main class

```
public class Main {
            private static final int noOfThreads=6;
            private static final int processorLoad=1000000;
            public static void main(String args[]){
                        Window win=new Window(noOfThreads);
                    for(int i =0; i<noOfThreads; i++){
                              new Fir(i,i+2,win,processorLoad).start();
                        }
              }
}
```

Code sequence 4: Window class

```
import java.util.ArrayList;
import javax.swing.JFrame;
import javax.swing.JProgressBar;

public class Window extends JFrame{
            ArrayList<JProgressBar> bars=new ArrayList<JProgressBar>();
           public Window(int nrThreads) {
                        setLayout(null);
                        setSize(450,400);
                        setDefaultCloseOperation(EXIT_ON_CLOSE);
                        init(nrThreads);
                    this.setVisible(true);
             }
            private void init(int n){
                     for(int i=0 ;i<n; i++){
                                  JProgressBar pb=new JProgressBar();
                                  pb.setMaximum(1000);
                                  pb.setBounds(50,(i+1)*30,350,20);
                                this.add(pb);
                                this.bars.add(pb);
                        }
              }
            public void setProgressValue(int id,int val){
                        bars.get(id).setValue(val);
              }
}
```

Code sequence: Fir class

```
public class Fir extends Thread {
            int id;
             Window win;
            int processorLoad;
             Fir(int id,int priority,Window win, int procLoad){
                      this.id=id;
                      this.win=win;
                      this.processorLoad=procLoad;
                      this.setPriority(priority);
             }
            public void run(){
                     int c=0;
                     while(c<1000){
```

```
                                    for(int j=0;j<this.processorLoad;j++){
                                                j++;j--;
                                        }
                                        c++;
                                    this.win.setProgressValue(id, c);
                                }
                        }
}
```

3.2 Application 2:

3.2.1 Requirements: Modify the application presented as an example in point 3.1 so that the refresh of the interface will be done through the *MVC design pattern*, using the *Observer-Observable* mechanism.

Develop the application specifications according to the model above.

Both the proposed design diagrams and the application code will be modified.

3.3 Application 3:

3.3.1 Requirements: To implement in *Java 2 SE* an application that has three threads of execution.

3.3.2 Specifications:

1. The application will be implemented in *Java 2 SE*.
2. The application has a graphical user interface in where three geometric shapes (for example, squares) will be added, initially arranged at the top of the window.
3. Each thread of execution is responsible for moving the squares, towards the bottom of the window, with a constant speed, randomly calculated (between a minimum and a maximum) for each square.
4. The threads of execution will be stopped when the geometric shapes exit the window perimeter. For stopping the threads, the *stop*() method will be defined, which will use a logical variable. This game should be resumed three times by crating other threads of execution for the new squares each time.

3.3.3 What is required:

a) class diagram;

b) activity diagram;

c) sequence diagram;

d) source code.

3.3.4 Testing if the application:

- opens the graphical interface;
- creates the 3 geometric shapes;
- moves the 3 geometric shapes'
- performs the stopping of the threads with the defined *stop*() method.

3.4 Application 4

3.4.1 Requirements: The application from point 3.3 will be modified so that at the bottom of the application a new geometric shape (a tank) will be added. The user will be able to move the tank using the keyboard, left or right and shoot shells by pressing space. The application will have a fourth thread of execution with the role of a supervisor. The tank is able to shoot the fallen squares which means killing the thread of that square. When the tank collides with one of the squares, all geometric shapes in the window will stop and an error message will be displayed in the window. In this application, the threads responsible for moving the squares will evolve in the loop (the moment a square comes out of the window, it will resume its initial position). The user will be able to resume the game three times. To make the game more interesting, a score will be calculated. It will be calculated by summing the shot squares successfully.

3.4.2 What is required:

a) class diagram;

b) activity diagram;

c) sequence diagram;

d) source code.

3.4.4 Testing if the application:

- opens the graphical interface;
- the new geometric shape is added;
- displays the error message when the figures collide;
- restart the game from its original state;
- can play the game three times;
- calculates the score correctly.

4. Verification of knowledge

1. Define the *thread of execution* concept.
2. What is the difference between a *thread of execution* and a *process*?
3. What are the techniques that can be implemented in *Java*? In what situations each one used?
4. What is the method that starts a thread of execution?
5. What is the method that contains the logic (sequence of instructions) of the thread of execution?
6. What does the priority of a thread refer to? How many priority levels are there in *Java*? What is the default priority of a thread?
7. Name at least 4 methods defined in the *Thread* class.