Java multithreading

Objectives:

- understanding the concepts related to execution threads;
- embedding threads into specific applications;

Execution threads

Concurrent programming or *multithreading* is a mechanism considered *lightweight* that represents the ability of a program to execute multiple code sequences from that program at the same time. Such a sequence of code is called *execution thread*.

Due to the possibility of creating multiple threads, a Java program can execute several tasks simultaneously, such as animating an image, playing a sound file, communicating with a server, etc.

In order for the Java programmer to be able to develop basic multithreading applications, the Java language offers two classes and a Java interface in the default `java.lang` package:

- interface `java.lang.Runnable`
- class `java.lang.Thread`
- class `java.lang.ThreadGroup`

The `Thread` class and the `Runnable` interface provide support for implementing and working with threads as separate entities. The `ThreadGroup` class allows creating groups of threads in order to treat them in a unitary way.

The executor-based mechanism is included in the `java.util.concurrent` package and was included in the language since Java SE5.

The programmatic control of the execution threads is based on the implementation of a special method named `public void run()` that is called by the JVM in order to run the specific instructions mentioned by the programmer.

Interface `java.lang.Runnable`

One way to create a thread is to use the `Runnable` interface.
The `Runnable` interface declares only one method, namely:

```
public void run();
```

Any class that uses (implements) this interface must define the implementation of the above mentioned method, thus defining the behavior of the thread.

Class `java.lang.Thread`

Defining a class that inherits the `Thread` class is another way to create a thread. The main member methods of the `Thread` class will be presented below.

Constructor methods

- `Thread()`

=> creates a new thread object

- `Thread(Runnable target)`

=> creates a new thread object

- `Thread(Runnable target, String name)`

=> creates a new thread object, having the name received as parameter

- `Thread(String name)`

=> creates a new thread object, having the name received as parameter

- `Thread(ThreadGroup group, Runnable target)`

=> creates a new thread object and associates it with the group received as a parameter

- `Thread(ThreadGroup group, Runnable target, String name)`

=> creates a new thread object with the name received as a parameter and associates it to the group received as a parameter

- `Thread(ThreadGroup group, Runnable target, String name, long stackSize)`

=> creates a new thread object with the name received as a parameter, associates it with the group received as a parameter and has the stack size received as a parameter

- `Thread(ThreadGroup group, String name)`

=> creates a new thread object with the name received as a parameter and associates it to the group received as a parameter

Main member methods

- `static int activeCount()`

=> returns the number of active threads in the current group of threads

- `void checkAccess()`

=> determine whether the current running thread has permission to modify this thread

- `static Thread currentThread()`

=> returns a reference to the current thread

- `void destroy()`

=> destroy this thread unconditionally

- `static void dumpStack()`

=> prints the current thread stack

- `static int enumerate(Thread[] tarray)`

=> copies into the thread array received as a parameter all the active threads from the current thread group and its subgroups

- `ClassLoader getContextClassLoader()`

=> returns the *ClassLoader* of the current thread

- `String getName()`

=> returns the name of the current thread

- `int getPriority()`

=> returns the priority of the current thread

- `ThreadGroup getThreadGroup()`

=> returnează grupul de care aparţine thread-ul curent

- `static boolean holdsLock(Object obj)`

=> returns *true* if the current thread has the lock of the specified object

- `void interrupt()`

=> interrupts the current execution thread

- `static boolean interrupted()`

=> tests if the current thread has been interrupted

- `boolean isAlive()`

=> tests if the current thread is active

- `boolean isDaemon()`

=> tests if the current thread is of a *daemon* type

- `boolean isInterrupted()`

=> tests if the current thread has been interrupted

- `void join()`

=> waits for the current thread to be finished

- `void join(long millis)`

=> waits no more than the number of milliseconds specified to finish the current execution thread

- `void join(long millis, int nanos)`

=> waits no more than the number of milliseconds and nanoseconds specified to finish the current execution thread

- `void resume()`

=> deprecated method, resumes the execution of the current thread

- `void run()`

=> if the thread was constructed using a *Runnable* object, the *run()* method of that object is called; otherwise, it does nothing (unless is overridden)

- `void setContextClassLoader(ClassLoader cl)`

=> sets the *ClassLoader* of the current execution thread

- `void setDaemon(boolean on)`

=> marks the current thread as *daemon* or user type

- `void setName(String name)`

=> sets the name of the current execution thread

- `void setPriority(int newPriority)`

=> sets the priority of the current thread

- `static void sleep(long millis)`

=> interrupts the current execution thread for at least the number of milliseconds mentioned

- `static void sleep(long millis, int nanos)`

=> interrupts the current execution thread for at least the number of milliseconds + nanoseconds mentioned

- `void start()`

=> starts the current execution thread; this method calls the *run()* method of the current thread

- `String toString()`

=> returns the textual representation of the current thread

- `static void yield()`

=> forces the current thread to temporarily cede access to the processor to allow other threads to run

Class `ThreadGroup`

This class is used to allow the grouping and unitary treatment of several threads.

Constructor methods

- `ThreadGroup(String name)`

=> creates a new group of threads, with the name received as a parameter

- `ThreadGroup(ThreadGroup parent, String name)`

=> creates a new sub-group of threads, having as parent the group received as a parameter; the name is also received as a parameter

Main member methods

- `int activeCount()`

=> returns the number of active threads in the current group

- `int activeGroupCount()`

=> returns the number of active subgroups in the current group

- `void checkAccess()`

=> determines whether the current thread has permission to modify this group of threads

- `void destroy()`

=> destroys the group of threads and all possible sub-groups

- `int enumerate(Thread[] list)`

=> copies into the thread array received as parameter all the active threads from the current group and from any sub-groups

- `int enumerate(ThreadGroup[] list)`

=> copies into the thread group array received as a parameter all the active sub-groups of the current group

- `int getMaxPriority()`

=> returns the maximum priority of the threads contained in the current group

- `String getName()`

=> returns the name of the thread group

- `ThreadGroup getParent()`

=> returns the parent of the current thread group

- `void interrupt()`

=> interrupts all threads in the current group

- `boolean isDaemon()`

=> tests whether the current group of threads is of *daemon* type

- `boolean isDestroyed()`

=> tests if the current group of threads is destroyed

- `void list()`

=> prints (to the console) information related to the current group of execution threads

- `boolean parentOf(ThreadGroup g)`

=> tests whether the current thread group is the parent of the group received as a parameter

- `String toString()`

=> returns textual information that describes the current thread group

- `void uncaughtException(Thread t, Throwable e)`

=> is a method that is called by the JVM when a thread in the current group of threads stops because of an un-flagged exception.

Creating a thread by using the interface `Runnable`

- creating a class that implements the interface `Runnable`.

```
class MyRunnable implements Runnable{
      //implementarea clasei
}
```

- the class that implements the `Runnable` interface must define the `public void run()` method with the code that is intended to be executed:

```
public void run(){
      //implementarea codului specific  firului de execuţie
}
```

- an instance of the created class is defined:

```
MyRunnable myRunnable=new MyRunnable();
```

- an object from the `Thread` class is created using a constructor that has as a parameter the `Runnable` type object.

```
Thread thread=new Thread(myRunnable);
```

- the thread is started by calling the method `start()`

```
thread.start();
```

Creating a thread using the base class `Thread`

- a class derived from the class `java.lang.Thread` is created

```
class MyThread extends Thread{
      //implementarea clasei
}
```

- the method `public void run()` inherited from `Thread` is overridden in the derived class. This method must implement the specific code of the execution thread.

```
public void run(){
      //implementarea comportamentului specific al firului de execuţie
}
```

- the defined class is instantiated:

```
Thread thread=new MyThread();
```

- the instantiated thread is started by calling the `start()` method. Calling this method causes the JVM to create the context necessary for the thread and then calls the `public void run()` method automatically.

```
thread.start();
```

Execution thread states

A thread may be in one of the following states:

- *new* (newly created)
- *running* (in execution, the state to which all the execution threads aspire)
- *blocked* (waiting, asleep, suspended, blocked)
- *ready* (ready for execution, waiting to be run)
- *dead* (finished)

A thread that is in *the dead* state can no longer be restarted. This attempt is viewed as an execution error by launching the exception:

```
java.lang.IllegalThreadStateException
```

After a thread is finished, it continues to exist as a java object, and the methods in the class can be called through the *dead* object. The only attribute that is lost by termination is related to its execution thread behavior.

Thread priority

The execution of multiple threads on a single CPU is called *scheduling*. Java supports a very simple deterministic planning mechanism based on fixed priorities. The thread planning algorithm is based on the relative priority over other runnable threads.
Java defines three symbolic constants for establishing thread priorities:

- `public final static int MAX_PRIORITY; //10`

- `public final static int MIN_PRIORITY; //1`
- `public final static int NORM_PRIORITY; //5`

Synchronizing threads

- the possibility of simultaneous access to common resources (variables, methods)
- if at most one thread has access to a code sequence at a certain time <=> *mutual exclusion*
- *producer-consumer* processes that allow mutual exclusion and inter-communication (cooperation)

Mutual exclusion is implemented with ***synchronized*** instructions and blocks of code. It is correlated with the concept of ***lock (monitor)***.

Monitor = an object that ensures that a shared variable can be accessed at a given time by no more than one thread. The monitor, in addition to mutual exclusion, allows thread inter-communication through the methods ***wait()*** and ***notify()***.

- used in synchronization
- `wait()`: run state => blocked for an indefinite period of time
- `notify(), notifyAll()` => reactivating blocked threads waiting for a monitor

Executors

- `Executor`: simple interface that allows running processes associated with threads
- `ExecutorService`: (derived from `Executor`) adds process and executor lifecycle management facilities
- `ScheduledExecutorService`: (derived from `ExecutorService`) additionally allows future or periodic execution of processes

There are 2 major implementation variants:

• Anonymous Inner Class

```
taskList.execute(new Runnable( ) {
public void run() { doSomeTask(…);}
                 });
```

• Lambda expressions

```
taskList.execute(() -> doSomeTask(…));
```

Exemple

Ex. 1. Threads, standard Runnable

```
/*This thread extends Thread class. */
class MyThread extends Thread
{
      @Override
      public void run( ){   //do something
            System.out.println("MyThread running");
      }
}
/** This thread implements Runnable interface */
 class MyRunnable implements Runnable
{
      public void run( ){
      System.out.println("MyRunnable running");
      }
}

/* Starts two threads */
public class ThreadStarter
```

```
{
	public static void main(String[ ] args)
	{
	Thread thread1 = new MyThread( );
	Thread thread2 = new Thread(new MyRunnable( ));

		thread1.start( );
		thread2.start( );
	}
}
```

Ex. 2. Threads, Runnable, Executors

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class MyThread extends Thread{
	@Override
	public void run(){
		System.out.println("MyThread running");
	}
}


class MyRunnable implements Runnable{
	public void run(){
		System.out.println("MyRunnable running");
	}
}

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

		//instantiere fir de executie propriu
		Thread thread1 = new MyThread();

		//utilizare clasa proprie care implementeaza Runnable
		Thread thread2 = new Thread(new MyRunnable());

		thread1.start();

		thread2.start();

		//expresii lambda si thread-uri
		Runnable another_runnable = () -> {
			try {
				String name = Thread.currentThread().getName();
				System.out.println("Mesaj 1 " + name);
				TimeUnit.SECONDS.sleep(1);
				System.out.println("Mesaj 2 " + name);
			}
			catch (InterruptedException e) {
				e.printStackTrace();
			}
		};
```

```
            Thread thread3 = new Thread(another_runnable);
            thread3.start();

            //executori
            ExecutorService executor = Executors.newFixedThreadPool(10);
            for (int i = 0; i < 10; i++) {
                  Runnable thread = new MyRunnable( );
                  System.out.println("Executor: fir de executie nr. "+i);
                  executor.execute(thread);
            }
            executor.shutdown( );
            while (!executor.isTerminated( )) { }
            System.out.println("Toate firele de executie din executor sunt
terminate.");

      }
}
```

Ex. 3. Threads, Callable, Future

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

class FactorialCalculator implements Callable<Integer>
{
     private Integer number;
     public FactorialCalculator(Integer number) {
        this.number = number;
    }
     @Override
    public Integer call( ) throws Exception {
        int result = 1;
        if ((number == 0) || (number == 1)) {
            result = 1;
        } else {
            for (int i = 2; i <= number; i++) {
                result *= i;
                TimeUnit.MILLISECONDS.sleep(20);
            }
        }
        System.out.println("Result for number - " + number + " -> " + result);
        return result;
    }
}

public class CallableExample {
  public static void main(String[ ] args)
  {
 ThreadPoolExecutor executor =(ThreadPoolExecutor)Executors.newFixedThreadPool(2);
        List<Future<Integer>> resultList = new ArrayList<>();
        Random random = new Random();
        for (int i=0; i<4; i++){
           Integer number = random.nextInt(10);
           FactorialCalculator calculator  = new FactorialCalculator(number);
           Future<Integer> result = executor.submit(calculator);
           resultList.add(result);
```

```
          }
         for(Future<Integer> future : resultList){
            try {
                System.out.println("Future result is - " + " - " + future.get( ) +
"; And Task done is " + future.isDone( ));
                } catch (InterruptedException | ExecutionException e)
                    { e.printStackTrace(); }
          }
                 //shut down the executor service now
                 executor.shutdown( );
     } // main
   }// class
```

Ex. 4. Primary animation

```
import java.awt.*;
import javax.swing.*;
import javax.imageio.ImageIO;
import java.io.*;

public class Animation_01 extends JFrame implements Runnable{
       int indimg=0;
       int iterations = 1;
       boolean should_run = true;
       Thread anim=null;
       String image_names[]= {"picture_01.jpg","picture_02.jpg","picture_03.jpg"};
       Image images[]=new Image[image_names.length];

       MyCanvas canvas;
       JLabel label;

       Animation_01(){
            super("...");
            canvas = new MyCanvas();
            label = new JLabel("Iteration: "+iterations+" / 3");
            setLayout(new FlowLayout());
            add(label);
            add(canvas);

            for(int i=0; i<images.length; i++){
                 try {
                      images[i] = ImageIO.read(new File(image_names[i]));
                 } catch (IOException ex) {}
            }

            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setSize(400, 400);

            anim=new Thread(this);
            anim.start();
       }
       @Override
       public void run(){
            while (should_run){
```

```
            canvas.setImage(images[indimg]);
            canvas.repaint();

            try{
                  Thread.sleep(1000);
            }
            catch(InterruptedException e) {}

            indimg++;
            if (indimg==images.length){
                  indimg=0;
                  iterations++;
                  label.setText("Iteration: "+iterations+" / 3");
            }


            if(iterations>3){
                  should_run = false;
                  label.setText("Iteration: "+iterations+" / 3"+" TOO MUCH!");
            }

            } //while
      }
}

class MyCanvas extends Canvas{
      Image image;
      MyCanvas(){
            setSize(300, 200);
      }

      public void setImage(Image image){
            this.image = image;
      }

      public void paint(Graphics g){
            g.drawImage(image,0,0,this);
      }
}

class Test{
      public static void main(String... args){
            Animation_01 frame = new Animation_01();
            frame.setVisible(true);
      }
}
```

Ex.5 Synchronized methods

```
class Parentheses {
     //void display(String s)
     synchronized void display(String s)
            {
                  System.out.print("("+s);
                  try{
                        Thread.sleep(1000);
```

```
                }catch(InterruptedException e){
                        System.out.println("Interrupted");
                }
                System.out.print(")");
            }//display
      }//Par

class MyThread implements Runnable{
      String s1;
      Parentheses p1;
      Thread t;
      public MyThread(Parentheses p2, String s2){
            p1=p2;
            s1=s2;
            t=new Thread(this);
            t.start( ); }//cons
                  public void run( ){
            p1.display(s1); }
                        }//MyThread

public class Demo{
      public static void main(String args[ ]){
                  Parentheses p3=new Parentheses( );
            MyThread name1=new MyThread(p3, "Bob");
            MyThread name2=new MyThread(p3, "Mary");
                  try{
                        name1.t.join( );
                        name2.t.join( );
                  }catch(InterruptedException e){
                        System.out.println("Interrupted");
                  }

      }
}//Demo
```

Ex. 6. Threads synchronization, mutual exclusion, producer-consumer

```
//Oracle Producer-Consumer
public class ProducerConsumerTest {
    public static void main(String[ ] args) {
        CubbyHole c = new CubbyHole( );
        Producer p1 = new Producer(c, 1);//Producer1 puts using object c
        Consumer c1 = new Consumer(c, 1);//Consumer1 gets using object c
        p1.start( );
        c1.start( );
    }
}
class CubbyHole {
    private int contents;
    private boolean available = false;
```

```
    public synchronized int get( ) {
        while (available == false) {
            try {  wait( ); } catch (InterruptedException e) { }
        }//while
        available = false; notifyAll( );//notify to put other value
        return contents; //we get the value
       }//get
  public synchronized void put(int value) {
        while (available == true) {
            try {
                wait( );  } catch (InterruptedException e) { }
        }//while
        contents = value;//we put the value
        available = true; notifyAll( );//notify to get the value
    }//put
}//class
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
   public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    @Override
public void run( ) {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get( );
            System.out.println("Consumer #" + this.number
                                + " got: " + value);
        }
    }//run
}//class
class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number; }
@Override
    public void run( ) {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random( ) * 100));
            } catch (InterruptedException e) { }
        }
    }//run
}//class
```

Individual work

Complexity *

1. Write a Java application which contains a class which implements the Runnable interface. The class' constructor sets the name of the instantiated object. Also, there is a class variable which counts the number of instantiated objects from that class. The *run( )* method of the class will print the object's name for a number of times equal to the counter's value, each printing being delayed 1000 msec.

In a distinct class, create multiple threads built from separate objects of the previously described class. Analyze the results.

2. Write a Java application with a thread that generates 30 random numbers between 0 and 30. Another thread displays the area of the circles having as radiuses the values divisible by 3 generated by the first thread.

3. Write a class for determining a certain value from Fibonacci's array. The class has 2 methods, one for calculating and the other for displaying the desired value. Use a synchronized multithreading mechanism in which one process displays all the Fibonacci numbers smaller than the desired value computed by the other process.

4. Write a Java app which uses the synchronized method acces for mutual exclusion. Create 3 separate threads which simultaneously call methods to increment and decrement a separate class' class variable. Check if the results are what you expect them to be. Remove the synchronized blocks and reevaluate the results.

Complexity **

5. Write a Java app which computes the greatest common divisor for large numbers (>100.000). The app will continuosly read from the keyboard pairs of numbers and launch threads for each of the pairs. The results will be displayed in the console as soon as they are available.

6. Write a Java application with a thread that writes some information into a file while another thread reads the written data and displays it on the screen. Synchronize the threads.

7. There is an urn which contains 1000 losing tickets and 1 winning ticket. There are N people around the urn (N is read from the keyboard) which simultaneously extract tickets from it. The tickets are not placed back into the urn. When one person extracts the winning ticket, the game stops.

Alternatives: 1) the persons extract the tickets in a predefined order; 2) the tickets are placed back into the urn.