4e année IR 2019/20

Object-oriented Programming

Java Threads, Multi-threading and Concurrency Management

S. Yangui (INSA/LAAS)

Creating Java threads

In Java, there are *two* ways to create threaded programs:

1. Extend the `Thread` class:

```
class MyThread extends Thread {
      ......
   }
```

2. Implement the interface `Runnable`:

```
class MyThread implements Runnable {
      ......
   }
```

Either one of these two approaches may be used, sometimes it is not possible to extend the `Thread` class, because you must extend some other class. For instance, in programming applets, you must extend the `Applet` class. Remember that Java does not support multiple inheritance. In those cases, where it is not possible to extend the `Thread` class, you need to implement the `Runnable` interface.

In this lab, we are using the first approach, that extends the `Thread` class. To create new threads in this case, simply use:

```
MyThread t = new MyThread();
```

Starting and stopping Java threads

The `Thread` class has three primary methods that are used to control a thread:

```
public void start()
public void run()
public final void stop()
```

The `start()` method prepares a thread to be run; the `run()` method actually performs the work of the thread - it serves as the main routine for the thread; and the `stop()` method halts the thread. The thread *dies* when the `run()` method terminates or when the thread's `stop()` method is invoked.

The class that extends `Thread` must redefine the `run()` method of `Thread`. To start the thread actually running, you do not invoke the `run()` method. Rather, you invoke the `start()` method on your object:

```
t.start();
```

The `run()` method is automatically called at runtime as necessary, once you have called `start()`. To stop a thread t, you may call

```
t.stop();
```

However, it is recommended to never stop a thread using the `stop` method (deprecated because it is not thread-safe). Let a thread stop when the thread function returns, and determine when the thread function should return using boolean variables as necessary.

Question A: What are the two ways you can provide the implementation for a thread's run method?

Exercise 0:

Considering the following multi-threaded code, compile (manually) the following code and write down the expected output.

```
class MyThread extends Thread {

  public void run() {
    System.out.println("Hi there");
  }
}

public class TestThread0 {
  public static void main (String arg[]) {
```

```
    MyThread t1, t2;

    t1 = new MyThread ();
    t2 = new MyThread ();

    t1.start();
    t2.start();
  }
}
```

Naming Java threads

It is often useful to give different threads in the same class names, so you can tell them apart. The constructor that allows you to do that is:

```
public Thread(String name)
```

This is normally called from the constructor of a subclass (your own), like this:

```
class MyThread extends Thread {
      MyThread(String name) {
            super(name);
      }

      public void run() {
            System.out.println(this.getName());
            ...
      }
}
```

The `getName()` method of the `Thread` class returns the name of the thread. The following program now distinguishes the output of different threads:

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

            MyThread first   = new MyThread("First ...");
            MyThread second  = new MyThread("Second ...");
            MyThread third   = new MyThread("Third ...");
            first.start();
            second.start();
            third.start();
      }
}
```

Exercise 1:

Considering the following multi-threaded code, compile (manually) the following code and write down the expected output.

```
class MyThread extends Thread {

  public MyThread (String s) {
    super(s);
  }

  public void run() {
    System.out.println("Hello, I am "+ getName());
  }
}


public class TestThread {
  public static void main (String arg[]) {
    MyThread t1, t2;

    t1 = new MyThread ("Thread #1");
    t2 = new MyThread ("Thread #2");

    t2.start();
    t1.start();
  }
}
```

Exercise 2:

Write a program `TestThreadMany.java` that takes a positive integer n from the command line and creates n threads that print out their own name. Here is a sample (expected) execution:

```
bash$ java TestThreadMany 4

Hello, I am Thread #1
Hello, I am Thread #2
Hello, I am Thread #3
Hello, I am Thread #4
```

Hint: Use the code TestThread.java (Exercise 1) as guidance.

Task scheduling

It is possible to code instructions to threads to schedule tasks for future execution in a background thread. These facilities are provided through the `Timer` class.

Exercise 3:

Convert `AnnoyingBeep.java` so that the initial delay is 5 seconds, instead of 0.

```
import java.util.Timer;
import java.util.TimerTask;
import java.awt.Toolkit;

/**
 * Schedule a task that executes once every second.
 */

public class AnnoyingBeep {
    Toolkit toolkit;
    Timer timer;

    public AnnoyingBeep() {
        toolkit = Toolkit.getDefaultToolkit();
        timer = new Timer();
        timer.schedule(new RemindTask(),
                       0,        //initial delay
                       1*1000);  //subsequent rate
    }

    class RemindTask extends TimerTask {
        int numWarningBeeps = 3;

        public void run() {
            if (numWarningBeeps > 0) {
                toolkit.beep();
                System.out.println("Beep!");
                numWarningBeeps--;
            } else {
                toolkit.beep();
                System.out.println("Time's up!");
                //timer.cancel(); //Not necessary because we call System.exit
                System.exit(0);   //Stops the AWT thread (and everything else)
            }
        }
    }

    public static void main(String args[]) {
        System.out.println("About to schedule task.");
        new AnnoyingBeep();
        System.out.println("Task scheduled.");
    }
}
```

Exercise 4:

Convert `AnnoyingBeep.java` to use the `scheduleAtFixedRate` method instead of `schedule` to schedule the task. Change the implementation of the `run` method so that if the run method is called too late for a warning beep (say, more than 5 milliseconds after it was scheduled to run), nothing happens--no beep and string are generated. (Hint: Remember your answer to Exercise 3.)

Exercise 5:

Change the main program of `TwoThreadsDemo.java` so that it creates a third thread, named "Bora Bora".

Compile (manually) and run the program again. Note that you will also need the SimpleThread.java program. Does this change your vacation destiny?

```
public class TwoThreadsDemo {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

Exercise 6:

Well-behaved threads voluntarily relinquish the CPU periodically and give other threads an opportunity to run. Rewrite the `SelfishRunner` class to be a `PoliteRunner`. Be sure to modify the main program in `RaceDemo.java` to create PoliteRunners instead of SelfishRunners.

```
public class SelfishRunner extends Thread {

    private int tick = 1;
    private int num;

    public SelfishRunner(int num) {
```

```
        this.num = num;
    }

    public void run() {
        while (tick < 400000) {
            tick++;
            if ((tick % 50000) == 0)
                System.out.println("Thread #" + num + ", tick = " + tick);
        }
    }
}

public class RaceDemo {

    private final static int NUMRUNNERS = 2;

    public static void main(String[] args) {
        SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];

        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i] = new SelfishRunner(i);
            runners[i].setPriority(2);
        }
        for (int i = 0; i < NUMRUNNERS; i++)
            runners[i].start();
    }
}
```

Exercise 7:

Compile and run `RaceDemo.java` and `SelfishRunner.java` on your computer. Do you have a time-sliced system?

```
public class RaceDemo {

    private final static int NUMRUNNERS = 2;

    public static void main(String[] args) {
        SelfishRunner[] runners = new SelfishRunner[NUMRUNNERS];

        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i] = new SelfishRunner(i);
            runners[i].setPriority(2);
        }
        for (int i = 0; i < NUMRUNNERS; i++)
            runners[i].start();
    }
}

public class SelfishRunner extends Thread {

    private int tick = 1;
    private int num;
```

```
    public SelfishRunner(int num) {
        this.num = num;
    }

    public void run() {
        while (tick < 400000) {
            tick++;
            if ((tick % 50000) == 0)
                System.out.println("Thread #" + num + ", tick = " + tick);
        }
    }
}
```