Real Time Systems

Exercise 1: Tools and Threads

2020 Version

The aims of this exercise are:

- to become familiar with the programming tools that you will use during exercises and projects
- to learn how to use threads in Java

The exercise is developed for Linux (Fedora, Java8). However, the exercises can also be done on other platforms, e.g. Windows or Solaris.

Java Documentation

When programming in Java it is useful to always have a web-browser tab open that shows the JavaDoc documentation for Java.

1. Open a browser tab that shows the Java8 Platform API Specification.
2. Look up the `Thread` class and check which methods are available

It is also useful to have a browser tab open that shows the API for the local classes in se.lth.control.*

3. Open a new browser tab that shows the API for se.lth.control.* available here.
4. Study the documentation of the classes AnalogIn and AnalogOut.

Oracle's Java tutorial is a good source of information about Java.

5. Go to Oracle's Java tutorial. Those of you without any experience of Java are recommended to go through the following in the trail *Learning the Java Language*:
    - Object-oriented programming concepts
    - Language basics
    - Classes and Objects
    - Interfaces and Inheritance

Editors

Any plain text editor will work fine, but it is recommended that you use an editor with syntax highlighting and automatic indentation.

gedit/geany/Notepad++

All of these are easy to learn plain text editors with ordinary hotkeys that have some extra features for programmers. Notepad++ is only available on windows while the others are available for many platforms. Programs written in these editors need to be compiled and executed from the command line.

Eclipse/IntelliJ/Xcode

These are more powerful tools that contain your entire programming environment. However, learning and getting used to them can take more work and can be confusing for beginners. Also, the teaching assistants might not be able to help you with problems related to your specific setup.

Threads

This exercise involves writing a simple Java program, `Periodic`, that uses *threads*. It is essential that you understand the meaning of all words that are indicated by *italic* letters.

Specification

The program will start a number of threads that execute periodically. The threads simply print their period on standard output once each period as below:

```
> java Periodic 500 2000
500, 2000, 500, 500, 500, 500, 2000, 500, 500, 500, 500, 2000, ...
```

The program starts one thread per input argument, with the period in milliseconds specified by that argument.

6. Create a new file `Periodic.java`. This is the file for your program source code. Declare the *class* `Periodic` as

```
public class Periodic extends Thread {
}
```

The *class* is declared to *extend* the `Thread` class, so that *instances* of the class may run as threads. Another way to express this is that `Periodic` is a *subclass* of `Thread`. In this program we want to be able to specify a period for each thread, i.e. each instance of `Periodic`. Therefore we add an *instance variable* to hold the period for each `Periodic` instance:

```
private int period;
```

7. Compile the class from a terminal with the command

```
> javac Periodic.java
```

If the compilation returns with errors or warnings you should check your code. If running outside Lab A, also make sure that your Java environment is setup correctly.

Declaring a Java program

To be able to run the class as a program we need to define the `main()` method:

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

The `main()` method is called once by the *JVM* when running a class as a program.

8. Compile the class again, and then run it as a program with the command

```
> java Periodic
```

Of course not much happens yet. Actually no thread was created when we ran the program, because no instance of the class was created.

Creating a class instance

To create a class instance, or an object, we need to call a *constructor* method of the class. When we create a `Periodic` instance we also want to set the period of that instance. Therefore a constructor method is defined as

```
public Periodic(int period) {
    this.period = period;
}
```

Now let `main()` call the constructor method:

```
Periodic p = new Periodic(1000);
```

9. Compile and run the program.

A `Periodic` object (which is also a `Thread` object) with period 1000 ms was created when we ran the program, and a *reference* to that object was stored in the variable `p`. But still nothing visible happened, because we have not specified what code the thread should execute, neither have we started the thread.

Running threads

The code that a running `Thread` object executes is defined in the `run()` method. The `run()` method of the `Thread` class is defined as empty. To include our own code we *override* the `run()` method in our subclass `Periodic`. We want the object to print its period on standard output. Therefore we define it as

```
public void run() {
    System.out.print(period);
    System.out.print(", ");
}
```

To start the thread we call the `start()` method of the `Thread` class after the object creation in `main()`, i.e.

```
p.start();
```

10. Compile and run the program.

Now we are getting somewhere! The `Periodic` thread printed its period on the standard out stream, but only once. This is because the `run()` method terminated after printing once, and as a result the thread stopped. To make the thread print periodically we need to add a loop and make it wait `period` milliseconds before printing the next. Let us use a while loop:

```
while (!Thread.interrupted()) {
    ...
}
System.out.println("Thread stopped.");
```

This makes the thread run until the thread is interrupted (requested to stop). To make the thread wait for a certain time we call the static `sleep()` method of the `Thread` class.

```
try {
    while (!Thread.interrupted()) {
        ...
        Thread.sleep(period);
    }
} catch (InterruptedException e) {
    // Requested to stop
}
System.out.println("Thread stopped.");
```

Since `Thread.sleep()` may throw an *InterruptedException* if the sleep is interrupted, we need to use the `try/catch` construction. Here it encloses the while loop to make sure that the thread is stopped if an `InterruptedException` is caught (if an `InterruptedException` is thrown the Thread's interrupted status is cleared). If we would have put the `try/catch` around the `Thread.sleep()` only, then the loop would not exit, and interrupting the thread would not cause it to exit.

11. Make `Periodic` print its period periodically. Compile and run.

The next step is to create multiple threads with different periods to run in parallel.

12. Create and start some other `Periodic` objects from the `main()` method as above. Compile and run.

[Expand solution]
Periodic.java:

```
public class Periodic extends Thread {
    private int period;

    public Periodic(int period) {
```

```
        this.period = period;
    }

    public void run() {
        try {
            while (!Thread.interrupted()) {
                System.out.print(period);
                System.out.print(", ");

                Thread.sleep(period);
            }
        } catch (InterruptedException e) {
            // Requested to stop
        }
        System.out.println("Thread stopped.");
    }

    public static void main(String[] args) {
        Periodic p = new Periodic(1000);
        p.start();

        new Periodic(2000).start();
    }
}
```

Reading program arguments

The last step is to read the input arguments to the program and start the corresponding threads. The program input arguments are delivered to the `main()` method in the `args` argument, as an array of `String` objects. The method `Integer.parseInt()` converts a `String` to an `int`. The `for` construction can be used to loop through an array, e.g. `for (String arg : args)`.

13. Loop through the program input arguments and construct the corresponding `Periodic` threads. Try to run the program with some different input arguments, e.g.

```
> java Periodic 500 1000
```

See if you can find situations when the output on the screen is corrupted. The screen is a shared resource. In Exercise 2 you will learn how to ensure that shared resoures are accessed under mutual exclusion.

14. How many threads are involved in the execution of the above command? Use the method `Thread.activeCount()` to verify your answer.

[Expand solution]
Periodic.java:

```
public class Periodic extends Thread {
    private int period;

    public Periodic(int period) {
        this.period = period;
    }

    public void run() {
        try {
            while (!Thread.interrupted()) {
                System.out.print(period);
                System.out.print(", ");

                Thread.sleep(period);
            }
        } catch (InterruptedException e) {
            // Requested to stop
        }
        System.out.println("Thread stopped.");
    }

    public static void main(String[] args) {
        for (String arg : args)
```

```
            new Periodic(Integer.parseInt(arg)).start();

        System.out.print("(Number of active threads = " + Thread.activeCount() + "), ");
    }
}
```

The number of threads involved in the execution of the command is three. The program starts two `Periodic` threads, and there is one Main thread executing the `main` method. The main thread is started by the JVM.

Creating threads by implementing `Runnable`

A drawback with creating threads by extending the `Thread` class is that it is not possible for the class to be a subclass of some class other than `Thread`, e.g. of some user-defined class. The reason for this is that Java only supports *single inheritance*, and not *multiple inheritance*. An alternative way is to specify that the class *implements* the `Runnable` *interface*:

```
public interface Runnable {
    public void run();
}
```

15. Define a new class `Base` in a new file. The class may be empty.
16. Create a new class `PeriodicRunnable` in a new file. It should extend `Base` and implement `Runnable`. It should have the same functionality as `Periodic`.

Since instances of `PeriodicRunnable` are no longer instances of `Thread`, starting them is slightly different:

```
PeriodicRunnable m = new PeriodicRunnable(...);
Thread t = new Thread(m);
t.start();
```

[Expand solution]
PeriodicRunnable.java:

```
public class PeriodicRunnable extends Base implements Runnable {
    private int period;

    public PeriodicRunnable(int period) {
        this.period = period;
    }

    public void run() {
        try {
            while (!Thread.interrupted()) {
                System.out.print(period);
                System.out.print(", ");
                Thread.sleep(period);
            }
        } catch (InterruptedException e) {
            // Requested to stop
        }
        System.out.println("Thread stopped.");
    }

    public static void main(String[] args) {
        for (String arg : args) {
            PeriodicRunnable p = new PeriodicRunnable(Integer.parseInt(arg));
            Thread t = new Thread(p);
            t.start();
        }
    }
}
```

Internal threads

A third possibility is to have threads as *inner classes*, i.e. classes that are defined within another class. The inner thread classes typically extend `Thread`.

17. Create a new class `PeriodicWithInnerThread` that extends `Base`. Inside the class definition you should define the inner class `PeriodicThread` by extending `Thread`. Use a field in `PeriodicWithInnerThread` to reference an instance of the inner `PeriodicThread` class. The functionality should be the same as before.

Hint: You cannot create an instance of an inner class without first creating an instance of the outer class.

[Expand solution]
PeriodicWithInnerThread.java:

```
public class PeriodicWithInnerThread extends Base {
    private int period;
    private PeriodicThread t;

    public PeriodicWithInnerThread(int period) {
        this.period = period;
        t = new PeriodicThread();
    }

    public void start() {
        t.start();
    }

    private class PeriodicThread extends Thread {
        public void run() {
            try {
                while (!Thread.interrupted()) {
                    System.out.print(period);
                    System.out.print(", ");
                    Thread.sleep(period);
                }
            } catch (InterruptedException e) {
                // Requested to stop
            }
            System.out.println("Thread stopped.");
        }
    }

    public static void main(String[] args) {
        for (String arg : args) {
            PeriodicWithInnerThread p = new PeriodicWithInnerThread(Integer.parseInt(arg));
            p.start();
        }
    }
}
```

An advantage with inner classes is that all variables and methods visible in the *outer class* can be directly accessed from the inner class. Another advantage is that it is possible for an object to contain multiple execution threads.

It is also possible to let the inner class be an *anonymous class*, i.e., a class with no name. Using this approach the new class `PeriodicWithAnonymousThread` would contain the following code:

```
public class PeriodicWithAnonymousThread extends Base {
    ...

        Thread t = new Thread() {
            public void run() {
                // Code to be executed
            }
        };

    ...
}
```

18. Study the class `PeriodicWithAnonymousThread` available in the solution below. Make sure that you understand it.

[Expand solution]
PeriodicWithAnonymousThread.java:

```
public class PeriodicWithAnonymousThread extends Base {
    private int period;
    private Thread t;

    public PeriodicWithAnonymousThread(int per) {
        period = per;
        t = new Thread() {
            public void run() {
                try {
                    while (!Thread.interrupted()) {
                        System.out.print(period);
                        System.out.print(", ");
                        Thread.sleep(period);
                    }
                } catch (InterruptedException e) {
                    // Requested to stop
                }
                System.out.println("Thread stopped.");
            }
        };
    }

    public void start() {
        t.start();
    }

    public static void main(String[] args) {
        for (String arg : args) {
            PeriodicWithAnonymousThread p = new PeriodicWithAnonymousThread(Integer.parseInt(arg));
            p.start();
        }
    }
}
```

Drawback with anonymous classes are that they can only be instantiated where defined and the code might become less readable. Anonymous classes are often used to implement event listeners in Swing.

Thread Priorities

All threads have a priority. In Java, a higher number means higher priority. If no priority is set the thread gets a default priority.

19. Modify the code of `Periodic` so that it prints out its priority before entering the while loop.
20. Do the same thing for `PeriodicRunnable`. This must be done in a slightly different way. (Hint: Use the static method `Thread.currentThread()`)

Solution shown below.

In real-time applications the priorities of the threads are very important. Investigate how you change the priorities of a thread.

21. Modify `Periodic` so that it executes at a priority that is one above the base priority.
22. Do the same thing for `PeriodicRunnable`.

[Expand solution]
Periodic.java:

```
public class Periodic extends Thread {
    private int period;

    public Periodic(int period) {
        this.period = period;
    }

    public void run() {
        System.out.println("Priority = " + getPriority());
        setPriority(getPriority() + 1);
```

```
        try {
            while (!Thread.interrupted()) {
                System.out.print(period);
                System.out.print(", ");

                Thread.sleep(period);
            }
        } catch (InterruptedException e) {
            // Requested to stop
        }
        System.out.println("Thread stopped.");
    }

    public static void main(String[] args) {
        for (String arg : args)
            new Periodic(Integer.parseInt(arg)).start();
    }
}
```

PeriodicRunnable.java:

```
public class PeriodicRunnable extends Base implements Runnable {
    private int period;

    public PeriodicRunnable(int period) {
        this.period = period;
    }

    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("Priority = " + t.getPriority());
        t.setPriority(t.getPriority() + 1);

        try {
            while (!Thread.interrupted()) {
                System.out.print(period);
                System.out.print(", ");
                Thread.sleep(period);
            }
        } catch (InterruptedException e) {
            // Requested to stop
        }
        System.out.println("Thread stopped.");
    }

    public static void main(String[] args) {
        for (String arg : args) {
            PeriodicRunnable p = new PeriodicRunnable(Integer.parseInt(arg));
            Thread t = new Thread(p);
            t.start();
        }
    }
}
```

Thread Models

Java has two different thread models. In the *green-thread model* the virtual machine (VM) is responsible for the scheduling of the Java threads, which only exist as abstractions inside the VM. In the *native-thread model* the Java threads are mapped upon the threads of the underlying operating system, i.e. the OS threads. The scheduling of the Java threads is then subject to the underlying scheduling of the threads by the operating system.

The green-thread model is the standard model for reference implementations of Java. Since it does not require any thread support in the operating system, it is very portable. However, for efficiency reasons modern VMs such as the Linux HotSpot VM used in the course use the native-thread model.

In the course we use standard Java and pretend that it is a real-time language, although in reality it is not. However, we want our programs to behave in a way that functionally is the same as if the programs would have been written in some real-time language executing on some real-time operating system. For example, we do not want a low priority thread to preempt a high priority thread.

Consider the following example:

```
public class StarvationTest {
    public static void main(String[] args) {
        Thread t_low = new LowPrioThread();
        t_low.start();

        Thread t_high = new HighPrioThread();
        t_high.start();
    }

    private static class LowPrioThread extends Thread {
        public void run() {
            setPriority(7);
            System.out.println("Low priority thread started.");
            try {
                while (!Thread.interrupted()) {
                    System.out.println("Low priority thread executing");
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                // Requested to stop
            }
            System.out.println("Thread stopped.");
        }
    }

    private static class HighPrioThread extends Thread {
        private double sum;

        public void run() {
            setPriority(8);
            System.out.println("High priority thread started.");
            while (!Thread.interrupted()) {
                sum = sum + Math.random();
            }
            System.out.println("Thread stopped.");
        }
    }
}
```

This is an example where the programmer has made a design error. The high priority thread never releases the CPU. The result will be that the low priority thread is starved, it never gets the chance to execute. The solution is either to let the high-priority thread release the CPU, e.g. by doing a sleep() in the loop, or to give it lower priority than the low priority thread.

23. Compile and execute the above example. What is the result?

[Expand solution]

```
> javac StarvationTest.java
> java StarvationTest
High priority thread started.
Low priority thread started.
Low priority thread executing
Low priority thread executing
Low priority thread executing
...
```

The problem is that the two OS threads used in the native-thread model to execute the two Java threads have the same priority and use round-robin time-slicing. The first OS thread executes the high priority thread. After a while its time slice expires, and the next OS thread starts executing. It chooses the next available Java thread, the low-priority thread, and starts executing it.

This behaviour is **NOT** the behaviour that we want on a uni-processor machine. It makes it difficult to detect errors caused by poorly written code. The user believes that everything is correct, even when this is not the case.