Lab 1: Introduction to the Java Threading API

Aims: The aims of this lab session are to:

- become familiar with Java API threads and threading, and to create some simple multi-threaded applications,
- explore issues associated with thread scheduling, thread priority, and thread scheduling policy.

Thread Creation

In this section we will begin by examining the two low-level techniques for constructing and running multiple threads in Java applications. Note that in practice we will sometimes prefer to use higher-level thread management techniques (such as thread Executors that we shall see next week), so today is about understanding the basics of how threads work rather than how we would necessarily create Threads when creating efficient programs.

Thread Creation by implementing the Runnable interface

To create a thread in Java we can create a class to contains the task that we want the thread to run by implementing the **`Runnable`** interface, and then pass an object of our Runnable class as a parameter to a **`Thread`** object.

Remember that, in Java, an **interface** specifies a required set of methods that a class needs to become a particular type of object. It is up to the implementing class to define the code within the methods declared in the interface. If we want to implement `Runnable` we only have to define the **`run()`** method.

In the following example, for convenience, we're defining the new `MyRunnable` class as a **nested inner class** within the application that's going to use it. Doing this means that we don't have to create a separate class file (although we could if we wished, and would do if we were planning to use it in several files rather than just one place).

Having developed a class which implements `Runnable`, we can create an object of class `Thread` and associate it with the `Runnable` object. To *run* the code, invoke `start()` on the thread, as in the example code below.

```
package lab01;
/* Thread Creation using a Runnable for the thread Task */

public class MainAppPattern1 {

  public static void main(String[] args) {

    MyRunnable r = new MyRunnable() ;    /* create a MyRunnable object */
    Thread t0 = new Thread(r, "t0") ;    /* create a Thread object and associate
                                          * it with the runnable object */
    t0.start() ;           /* starts concurrent execution of thread t0 */
    /* this has started a new thread, running concurrently with main method */
  }

  static class MyRunnable implements Runnable {
    // supply the public void run() method required by Runnable interface
    @Override public void run() {
      /* put code to be executed when a thread starts in the run() method.
      * This example prints the thread name and a count */
      String tname = Thread.currentThread().getName() ;
      int count = 0;
      for (;;) { // empty for loop creates an infinite loop, or use while(true)
        System.out.println("This thread is " + tname + " " + count++) ;
        // try{Thread.sleep(10);}
        // catch(InterruptedException e){/* ignore */}
} } } }
```

- Applying `start()` method to the `Thread` object calls the `run()` method of its associated `Runnable` object.
- The code defined by the `Runnable` object's `run()` method **IS** the thread's task.
- The thread's `run()` code is executed concurrently with `run()` code of any threads started at the same time.

Thread Creation by extending the Thread class

We can also create a "threadable" class by inheriting from the `Thread` class and overriding its empty inherited `run()` method. This creates a sub-class of Thread that has its own bespoke task. Again we are defining the thread class as an nested inner class within the application. This pattern takes the form shown in the sample code below:

Then, in an application, create a new "threadable" object based on this class and apply the `start()` method to it.

```
package lab01;

/* Thread Creation using a specialised Thread with overridden run method
 * The Thread class is declared in java.lang, so no need to import it */
/* There is no significance to class name MyThread, use any name you like */

public class MainAppPattern2 {
  public static void main(String[] args) {
    // create a MyThread object i.e. an object which can be run as a thread
    MyThread t0 = new MyThread("t0") ;
    // now start the concurrent execution of thread t0
    t0.start() ;
    // started a new thread. What is it running concurrently with?
  }

  static class MyThread extends Thread {

    public MyThread(String name) {
      super(name) ; // Thread super-class constructor to initialise thread name
    }

    /* override inherited Thread's empty method run()
     * the method name run() is important here; whenever a thread starts
     * it expects to call a method with the signature public void run() */
    @Override public void run() {
    /* put code to be executed when this thread starts in the run() method.
     * This example just prints the thread name and a count. */
      String tname = Thread.currentThread().getName() ;
      int count = 0;
      for (;;) {
        System.out.println("Thread name: " + tname + " count: " + count++) ;
        try{Thread.sleep(10);}
        catch(InterruptedException e){/* ignore */}
      }
    }
  }
}
```

- The `start()` method in turn calls the overridden `run()` method.
- The code defined by the `Thread` object's `run()` method **IS** the thread's task.
- The thread's `run()` code is executed concurrently with `run()` code of any threads started at the same time.

Exercise 1

Aim: Code an example that creates a Thread by implementing **`Runnable`** and submitting to an empty **`Thread`**

a) Using an Integrated Development Environment (IDE) (e.g. VS Code), create a new project folder called **`lab01`**. In the folder **`lab01`** folder create a source file called **`Ex1.java`**.

b) Implement **`Ex1.java`** using `MainAppPattern1` as a starting point from earlier in this lab sheet.

c) The main method of **`Ex1`** should create one MyRunnable object, and then submit the task to three separate Thread objects. It should then start each of the three threads so that they run concurrently i.e. `Ex1` should create and start three threads that repeatedly display their own names, using the `MyRunnable` object to give the threads their tasks.

d) Run `Ex1.java` and observe the program's run-time behaviour in the output.
Do all threads seem to get a FAIR and equal chance to make progress in their computation?

e) Uncomment the try-catch statement inside the for-loop of the run() method of MyRunnable.
Re-run the code. What effect does the sleep method have?

You can vary the length of the sleep period in the Thread.**sleep(10)** method in the run() method, to speed up or slow down the thread's output.

Note - to run an individual file in VS Code, you can right-click and use **Run Java**

Note that threads in Exercise 1, and several later exercises, will run indefinitely, so you will have to stop the Java process or else it will continue forever. You can stop the process by pressing Ctrl+c when the cursor is in the output terminal.

Exercise 2

Aim: Code an example that creates a Thread by extending **`Thread`**

(a) Start by creating a new class `Ex2.java` in the folder **`lab01`**, with the code from **`MainAppPattern2`** earlier in this lab sheet as a starting point.

`Ex2` should create three MyThread objects with different names (i.e. each thread should repeatedly display its own name). It should then start each thread so that they run concurrently i.e. `Ex2` should create and start three threads, using the `MyThread` that have overridden run methods that contain their tasks.

(b) Run `Ex2.java` and observe the resulting program output.

Do all threads seem to get a FAIR and equal chance to make progress in their computation?

You can vary the length of the Thread.**sleep(10)_** in the run() method to speed up or slow down the thread's output.

Note `Ex1` and `Ex2` should behave the same, since as far as threading is concerned, there is no difference in performance or functionality between *Runnable in Thread* or *Thread with run method* approaches to thread construction.

However, use of the Java interface `Runnable` is sometimes preferred because we can *also* inherit from any other class, at the same time that we implement `Runnable`. This approach gives us more opportunity to reuse existing code in our designs.)

Exercise 2 continued

In Ex2, experiment with the following changes:

(c) Make the threads daemon threads (i.e. background threads) before applying start e.g.

```
t1.setDaemon(true); // and same for other threads
```

What happens now, and why?

(d) Leave the threads as daemon threads, but pause the main method using e.g.

```
Thread.sleep(1000L)
```

after the threads are started. Note that sleep() it will need to be wrapped inside a try-catch statement because it can throw an exception that must be handled.

What happens now, and why?

Experiment with the value in that sleep method, e.g. try 1L, 10L, 100L or 10000L instead of 1000L

(e) Return the threads to non-daemon threads (by commenting out the lines introduced in (c)) but now adapt the loop in the run() method of **MyThread** so that it is controlled by whether the thread has been interrupted or not – to see the effect you'll also have to temporarily comment out the sleep

```
while(!interrupted()){ // rather than while(true){
  System.out.println("Thread name: " + tname + " count: " + count++) ;
  // try{Thread.sleep(1);}
  // catch(InterruptedException e){/* ignore */}
}
```

And after the sleep in the main method of the application invoke interrupt() on the threads. e.g.

```
t1.interrupt(); // and same for the other threads
```

What happens now, and why?

(f) Change the start() call of the first thread to a run() call instead e.g. `t1.run();`

What happens now, and why? Revert back to using start() once you've seen what run() did.

Before moving on to Exercise 3 find out more about the class level methods:

```
Thread.currentThread()
Thread.getName()
```

Refer to the JDK API online help (http://docs.oracle.com/javase/8/docs/api/). Look up the details for class `java.lang.Thread` (via **Thread** under All Classes in the API documentation) for full details of instance methods and other class features that are used in the following exercises.

Exercise 3

Aim: To directly compare a single-threaded application to a multi-threaded application with the same functionality.

(a) Consider the program `MultiTasks.java` below, which carries out multiple computational tasks in sequence (i.e. tasks are performed one after the other in a single thread of execution, i.e. in the main thread).

Create a new class, `MultiTasks.java` in **`lab01`**, and copy the code from the example below.

Run `MultiTasks` as a non-concurrent (i.e. conventional single-threaded) application and note the start and stop times and duration for each task, the final task, and for the main thread. You can vary the number of tasks (try e.g. 2, 4, 8, 16, 32, 64).

```
package lab01;

import java.time.Duration;
import java.time.LocalTime;

public class MultiTasks{

  public static final long MAX = 1000000000L ;
  public static final int NUMBEROFTASKS = 4;
  private static int tasksLeft;
  private static LocalTime mainstart;

  public static void main(String[] args){
    mainstart = LocalTime.now();
    System.out.println("Main thread started at " + mainstart);
    tasksLeft = numberOfTasks;

    for(int i = 0; i < NUMBEROFTASKS; i++){ task(i) ; }

    LocalTime finish = LocalTime.now();
    System.out.println("Main thread ended at " + finish + " after running for "
        +  Duration.between(mainstart,finish).toMillis() + " ms")  ;
  }

  public static void task(int id){
    LocalTime start = LocalTime.now();
    System.out.println("Task " + id + " started at " + start);
    long sum = 0 ;
    for (long  i = 0 ; i < MAX ; i++) { // this creates a time-consuming loop
         sum++ ;
    }
    LocalTime finish = LocalTime.now();
    System.out.println("Task "  + id + " ended at " + finish
        + " with sum = " + sum + " after running for "
        +  Duration.between(start, finish).toMillis() + "ms")  ;
    tasksLeft--;
    if(tasksLeft == 0 ){
        System.out.println("Last Task ended at " + finish
            + " total run time for " + numberOfTasks +  " tasks is "
            +  Duration.between(mainstart, finish).toMillis() + "ms")  ;
      }
  }
}
```

(b) Your job now is to convert the code from `MultiTasks` into a multithreaded application in which each of the tasks is executed concurrently in its own thread.

- Create a file **`Ex3.java`** that contains a main method with the same sequence of actions as the main method of `MultiTasks` above, except that the tasks are created and started as Threads, rather than as invocations of the task method.

  i.e. in the main method of Ex3, replace

  ```
  for(int i = 0; i < NUMBEROFTASKS; i++){ task(i) ; }
  ```

  with something like

  ```
  Thread[] tasks = new Thread[NUMBEROFTASKS];
  for(int i = 0; i < NUMBEROFTASKS; i++){
    tasks[i] = new Thread(new Task(i));
  }
  for(Thread task: tasks){ task.start(); }
  ```

  Note that we could have done this more compactly, by creating *anonymous* threads
  e.g.
  ```
  for(int i = 0; i < NUMBEROFTASKS; i++){
    new Thread(new Task(i)).start();
  // starts tasks in anonymous manner
  }
  ```
  But in later parts of the exercises we'll need to refer to individual tasks to change their properties, so storing them in an array will allow us to do that.

- Create a nested inner class within the file **`Ex3`** called **`Task`** that performs the same actions as the `task()` method from the class `MultiTasks` above.

- Implement **`Task`** so that it can be run in a separate thread. i.e. **`Task`** should either extends Thread or implements Runnable.

Run sequential **`MultiTasks`** and concurrent **`Ex3`** for different values of **`numberOfTasks`** and note the typical time per task, and the total time for all tasks for each task in the sequential and threaded versions of the application.

	Sequentially in MultiTasks.java		Concurrently in Ex3.java	
numberOfTasks	Typical time per task	Total time for all tasks	Typical time per task	Total time for all tasks
2	69ms	139ms	33ms	47ms
4	152ms	609ms	40ms	56ms
8	240ms	1922ms	46ms	83ms
16	372ms	5965ms	100ms	200ms
32	623ms	19920ms	160ms	340ms
64	1082ms	69309ms	240ms	555ms

(c) Explain the difference between the timings obtained by running the tasks in sequence and by running the tasks concurrently.

Can you infer how many processors the Java application has available to it?

To get Java to tell you how many processors there are on your machine, add the following line of code at the start of the start of the main method:

```
System.out.println("This machine has "
      + java.lang.Runtime.getRuntime().availableProcessors()
      + " processors available");
```

(d) In **`Ex3`** temporarily replace all the thread **`start`** calls with **`run`** calls, e.g.
Replace `t.start() ;` with `t.run() ;`
Are the tasks still running concurrently if you use **`run`** instead of **`start`**?

Thread Priority and Scheduling

In this section we shall explore issues associated with thread scheduling and priority, and through exercise 4 will investigate the priorities of the threads in a simple program.

In the previous exercises we saw evidence of concurrency. Each thread gets an opportunity to progress its computation in small amounts or "slices" of time. On a hardware platform that has a single Central Processing Unit (CPU) all threads make progress by being repeatedly "switched on and off" the CPU. In effect the CPU is shared between all threads.

The switching of threads is not done in an arbitrary manner. Rather, there is a decision-making process called SCHEDULING which:

- decides when the currently active thread should relinquish the CPU and
- decides which thread should be allowed to run on the CPU next.

A detailed explanation of the effect and behaviour of the scheduling process depends on:

- a thread attribute called PRIORITY and
- Java's built-in thread SCHEDULING POLICY.

Thread Priority

The `Thread` class has class constants that indicate the range of values used by Java to indicate the priority of threads:

- `Thread.MIN_PRIORITY` is a final constant declared in `java.lang.Thread`
- `Thread.MAX_PRIORITY` is a final constant declared in `java.lang.Thread`
- `Thread.NORM_PRIORITY` is a final constant declared in `java.lang.Thread`

Whenever a thread is created, by default, it inherits the SAME priority as its creating thread. It is possible to find out the priority of any thread by calling:

- `int getPriority()` an object method in class `java.lang.Thread`

Exercise 4

Aim: explore the priority property of threads.

Make a copy of **`Ex3.java`** and rename it as **`Ex4.java`**

Set `NUMBEROFTASKS` to 4.

(a) Add some System.out.println statements to the main method of **`Ex4.java`** in order to display the values of the class constants:

- `Thread.MIN_PRIORITY`
- `Thread.MAX_PRIORITY`
- `Thread.NORM_PRIORITY`

(b) Add some System.out.println statements to the main method of **`Ex4.java`** in order to display the priorities of each of the threads that are running (i.e. the main method and al the separate threaded tasks that it starts)

Hints: (1) for a thread named **t**, the object method **t**.getPriority() returns it's priority

(2) Thread.currentThread().getPriority() will return the priority of the "current" thread, so if this is run in the main method it will give the priority of the main thread.

Run the code to inspect the priorities, and to check that the run-time system FAIR to all threads?

(c) Change the priorities of the first two threads, using e.g. after the threads are created, but before the threads are started, include

```
tasks[0].setPriority(Thread.MIN_PRIORITY);
tasks[1].setPriority(Thread.MAX_PRIORITY);
```

so that thread 0 has its priority lowered, thread 1 has its priority raised

Run the file now and observe the total execution time for each thread.
Is the run-time system still equally FAIR to all threads? It probably is!

NOTE It is likely that with `NUMBEROFTASKS = 4`, the change in priority has no observable effect, because the number of threads is small, and more importantly is likely to be less than the number of available processors available to the JVM if you are working on a modern computer. This will be the case whenever the number of threads is less than the number of processing cores available to the JVM.

To get Java to tell you how many processors there are on your machine, add the following line of code at the start of the start of the main method:

(d) Change the value of `NUMBEROFTASKS` to e.g. 8, then 16 then 32 and re-run the program.
Is the run-time system still equally FAIR to all threads?
Are the timings for threads 0 and 1 significantly different to the others?

Exercise 5

Aim: explore the priority property of threads.

Make a copy of **`Ex3.java`** and rename it as **`Ex5.java`**

In Ex5 change the value of MAX to MAX = 1000000L (otherwise you'll wait a *long* time when running the code!

Adapt the code so that one of the threads relinquishes its slice of CPU time after each occasion that it performs the arithmetic addition to variable `sum` i.e. add the following if statement inside the for-loop inside the **`Task.java`** file

```
// ... in method run() of class Task ...
for (long  i = 0 ; i < MAX ; i++){
  sum++  ;
  if (id == 0) Thread.yield(); // ADD THIS NEW LINE OF CODE HERE
}
```

Observe and explain the run-time behaviour.
How long does thread 0 take to execute compared to the other threads.

Exercise 6

Consider the following **`Clock`** class, which can perform some useful "background" task, e.g. keeping track of and displaying the current time while all other threads perform their application task.

```
package lab01;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public class Clock extends Thread {
  private long interval ;
  private DateTimeFormatter time_format;

  public Clock(long time) {
```

```
    interval = time ;
    time_format = DateTimeFormatter.ofPattern("HH:mm:ss:SSSS");
    this.setPriority(Thread.MAX_PRIORITY) ;
    this.setDaemon(true) ;
  }

  @Override
  public void run() {
    while(true){
      System.out.println("Clock says: " + LocalTime.now().format(time_format)) ;
      try {
        Thread.sleep(interval) ;
      } catch (InterruptedException ex) {/* ignore exception */}
    }
  }
}
```

That is used by the following application:

```
package lab01;
public class Ex6 {

   public static final long MAX = 100000000L ;
   public static final int NUMBEROFTASKS = 4;

    public static void main(String[] args) {

      Clock clock = new Clock(10);
      clock.start();

      for(int i = 0; i < NUMBEROFTASKS; i++){
        Thread t = new Thread(new Task(i));
        t.start();
      }
    }

    static class Task implements Runnable{
        private int id;

        public Task(int id){ this.id = id; }

        @Override public void run(){
          long sum = 0 ;
          for (long  i = 0 ; i < MAX ; i++){
            sum ++  ;
            if (sum % 1000000 == 0) System.out.println("Thread " + id
                 + " is at " + sum);
          } // creates a loop but only prints once every 1000000 times
      }
    }
}
```

(a) Make a copies of **Ex6 and Clock**.
(HINT: simply construct a `Clock` object and start the thread withing Ex6 (set the time to be e.g. 10ms), before creating and starting all other "application" threads), and observe the behaviour of the application.

NOTE: a Java thread is either a **daemon** OR **non-daemon** thread. daemon threads have no termination rights, so a daemon thread **is forced to** die when all other non-daemon threads have terminated.

(b) Demonstrate that that the `Scheduler` thread really does "die" when all other application threads have terminated. HINT: comment out the line of code `schedulerThread.setDaemon(true);` in the Scheduler class and observer the difference in program behaviour.

Java Scheduling Policy

The Java scheduler is part of the Java Virtual Machine. It reschedules threads whenever the run-time system changes state, i.e. when a **rescheduling point** is reached.

Rescheduling involves:

PRE-EMPTING the currently active thread, [pre-empting means temporarily interrupting]
SELECTING another thread which is ready to execute, and
RESUMING execution of the other thread from the point at which it was previously pre-empted.

If rescheduling occurs frequently enough and involves all threads, the run-time system can create the illusion of pseudo-parallelism i.e. all threads seem to execute in parallel.

Rescheduling points occur whenever any thread changes state e.g.

- the active thread gets blocked (and is no longer ready to run),
  - e.g. when the thread requests some slow input/output operation to be performed
  - e.g. when the thread puts itself to "sleep"
- a blocked thread gets unblocked (and becomes ready to run),
  - e.g. when an input/output operation previously requested by the blocked thread is completed
  - e.g. when the blocked thread's period of "sleep" has elapsed
- a new thread is created and started,
- a thread completes its execution and terminates,
- one thread is forcibly stopped by another.

Scheduling involves a decision-making process: the scheduler has to choose the next thread to resume execution. Modern run-time versions (i.e. after Java 1.2) java adopts a particular scheduling policy with two properties:

Property 1: a higher priority thread will always pre-empt a lower priority thread,

Property 2: threads of equal priority get an equal share of available processor time.

There are a number of "tricks" that we can use that will influence and alter thread scheduling behaviour:

- do not create all threads with equal priority,
- force the currently active thread to block (e.g. put itself to sleep),
- force the currently active thread to relinquish the CPU i.e. do not wait to be pre-empted, let the scheduler give the remainder of this time-slice to another thread (e.g. yield)
- design our own java scheduler