Uttara InfoSolutions

www.uttarainfo.com

Threads Lab

1) Write a program to print 1 to 100 to monitor using a thread of execution.

Steps:

a) Create subclass of Thread called PrintThread and override run() with the for loop to print 1..100 to monitor

b) Create TestThread class with main()

c) In main(), create object of PrintThread and invoke start() on it. Before and after print main()->starting and main()->ending to the monitor from main().

Points to think about:

a) Do you know what happens when start() is invoked on a thread object?

b) What happens if you call start() more than once on the same object? Try it

c) What happens if you call run() on PrintThread object instead of start()? Is there any difference in the output here? What is the advantage then of creating threads?

d) What happens when you create 2 objects of PrintThread and invoke start() on them? Observe the output.

e) Give a name to each thread object you create of PrintThread by invoking setName("Ramu"), etc. In the run(), when you print SOP, invoke Thread.currentThread().getName() and contact with the count value. Do you understand what Thread.currentThread() returns and how we are getting the name of the running thread? Test it. Set the priorities to the 2 threads by invoking t1.setPriority(10) and t2.setPriority(1) and execute the program. Is this affecting the program execution in a deterministic manner? why not?

f) Execute the program 2-3 times. Are you getting the same output? Why not? What if we do can we get fairly ordered output?

g) Add Thread.sleep(500) in run() after the SOP inside the for loop. Embed all the statements in run() inside of try..catch. Now test it. Are you getting more ordered output now? Recollect what happens to the state of the 2 threads at runtime.
h) In the run(), create String str = null; and then invoke str.length(); This will raise a NullPointerException. Observe whether main thread will crash when this thread throws the exception? Why not?

2) Write a program to create the same printing 1..100 with printing the name of the thread to the monitor and Thread.sleep(500) in overridden run() in a class named MyJob that implements Runnable. Now in main() of TestThread2 class, create object of MyJob and then create 2 Thread objects and pass same MyJob reference as parameter to it. Set the names to the threads and invoke start() on both. What output do you see? Is there any difference in using the first approach or second approach to plugging in the job at runtime? Which is better and why?

3) Modify the job to accept a start int value as parameter, so that the counting starts from that value, 100 times instead of starting from 0.
Steps:

a) Create MyJob class with int val instance variable
b) Create a parameterised constructor to accept the int as parameter. Set the val instance variable value with that of the passed parameter. This will become the input to your job.
c) In run(), modify the for to become for(int i = val ; i < val+100 ; i++). Do you understand now how to pass parameters to jobs?
d) Create 2 different job objects, one with 10 and other with 1000. Give different names and then invoke them in 2 different threads of execution. Observe the output.

4) Create ParamJob class that implements Runnable and override run(). Create 1 instance variables -> int output, generate getter method for this. In run(), generate a random value, multiply it by 1000000 and store in output instance variable.
Create class TestParamJob -> main() ->

a) create ParamJob object (job ref), create Thread object (t1 ref), link job to Thread and invoke start().
b) In main(), call job.getOutput() and print the returned value to monitor. Observe what value will be printed out. Why?

c) Invoke t1.join() before the SOP in main(). Now run and see if you get the generated value. Do you understand how jobs can return values to invoker? Do you understand how join() method affects thread execution? ask doubts if you have any.

5) Write a program to i) take a path of a file as input and calculate how many letters are there in that file ii) take a number as input and generate a prime number that is greater than that. In main(), print both the results.
Steps:

a) Create LetterCounterJob class that implements Runnable. Provide a parameterised constructor to accept path and set it to instance variable. In run(), using a BufferedReader, read each character from the file, check if it is a char and increment a count variable (instance variable with getter as this is output from the job).
b) Create NextPrimeJob class that implements Runnable. provide a parameterised constructor to accept a long value (input) and set it to instance variable. In run(), generate a new random number, multiply it with 10000000L, check if the value is greater than the input number, then check if it is prime, if yes, store this to an output instance variable (with getter()).
c) In main(), create the 2 job objects, create 2 Thread with the jobs, start it, invoke t1.join(), t2.join() and then invoke job1.getCount() and job2.getPrimeNum() to print the outputs to monitor.
d) How to know we are taking lesser time to execute these jobs in multithreading when compared to sequential execution? In run() of both jobs,

```
long t1 = System.currentTimeMillis();
// implement the job code
long t2 = System.currentTimeMillis();
```

SOP(t2 - t1); => this will give you the number of milliseconds that took to perform this job. Now verify if the jobs are executing faster when compared to sequential execution.

6) Create a Logger Singleton class that can log to a file. Every time you ask the logger to log, it should log to the file in a new thread of execution. (See Logger.java and TestLogger.java example demo files)
Logger singleton implementation:

a) Create a private constructor
b) Create a private static Logger obj ref var

c) Create a public static Logger getInstance() method in which you do the following:

```
if(obj==null)
	obj = new Logger();

return obj;
```

d) Create a log() method to accept String data. Code it thus:

```
public void log(final String data)
{
	new Thread()
	{
		public void run()
		{
			BufferedWriter bw = new BufferedWriter(new
FileWriter(path,true);
			bw.write(new Date()+":"+data);
			bw.close();
			// write best practice IO code to do the above
		}
	}.start(); // I will explain how this works in class.
}
```

7) Create a Counter class with int count instance variable. Create incCount() method that increments the count and returns it. Create CountJob class that implements Runnable and holds Counter as an instance variable. In run(), invoke counter.incCount() and print the returned value. In main(), create 1 Counter object, 1 Job object with counter ref, 2 Threads with same job, start it and verify if you are getting same count value. Mark incCount() synchronised and then see the output. What was the problem? Do you understand how this fixes the problem? (See CounterJob.java and TestCounter.java example java code)

8) Create a Job to take a List<Integer> as input. In run(), add all the values in the list and find the average. Use BigDecimal to add the values to. Set BigDecimal instance variable as output from the job. Time the job. See BigDecimal javadoc and you can easily use this.