SYSC 3303 Real-Time Concurrent Systems
Lab #2

Getting Started

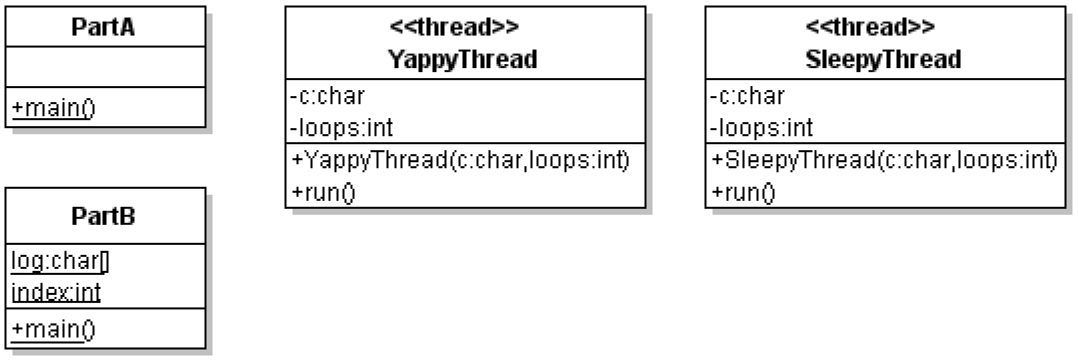
1. Reference: java.util.Thread. To learn about its API, google "Java API Thread".
2. When working with an IDE, please do not use JCreator or BlueJ. Please migrate to Eclipse or NetBeans.

Objective: Experiments with Java Threads

- Practical programming experience in Java's built-in threads
- Early experience in concurrency experiments: Using heuristics and delays to manifest concurrency issues.
- Early lessons in event logging in concurrent systems: Print versus memory logs.

Description:

Here's a class diagram of the code that you will be writing. The two main() classes will simply create the two thread classes: Yappy and Sleepy. The constructor for each of the two thread classes has two arguments: a unique character that will be used to identify the created thread object (i.e. a 1-letter name �Y� and �S�) and the number of loops the thread shall run. The lectures identified two methods for creating Java classes: you must use both methods, one each for each thread.



Heads Up: Please scan over the whole lab first. In doing so, you will see that you will end up writing two versions of the YappyThread and the SleepyThread classes, one for Part A (using printing) and one for Part B (using logging). You are free to manage your files and classes however you wish; this is not an OO course and this lab is a throwaway exercise. My imperative is to keep the focus on threads and logging. Some tips:

- The basic tenet of Java file management in which Java permits only one PUBLIC class per file. However, you can have other non-public classes in the file.
- In a project, you can only have one class named YappyThread.
- Suggestion: Create two projects, called PartA and Part B.
- You are then free to either:
  - Create three public classes (i.e. three files): PartA/B, YappyThread and SleepyThread, or
  - Create one public class (i.e. one file) called PartA/B and two non-public classes for the two threads.
- The key difficulty will be in PartB where the two thread classes will want to share the log (data) in the main class. There are different solutions. Any is fine for this throwaway exercise (but not in your assignments or project)
  - Make the log public and static
  - Provide public accessors for the log
  - Explore the Java language further by trying inner classes (an exercise in scope)
- For the log itself, please use arrays, rather than ArrayList (lighter, built-into language) but remember that arrays in Java must be allocated!

```
public class PartB
{
      public static char log[];
      public static void main(...) { }
} // Notice: Curly bracket ends the scope of PartA class

class Sleepy // Notice: No public!
{
}
�
class Yappy // Notice: No public!
{
}
```

Part A: Printing

In Part A, the two thread classes will both loop for the given number of times, and each time through the loop, it will print out its one-letter name.

- Please use print(), not println() to avoid lots of scrolling.
- Why a one-letter name? In this lab and every lab, when you print, **print out the minimal necessary to be understandable**. Long print messages affect the performance and clutter the screen. In this lab, you **must** just print the CHARACTER name of the thread ('y' for yappyand 's' for sleepy). If you choose to print out "This thread's name is Sleepy" (i.e. ignore my instructions), your lab will not be marked. Why such a big deal?� **It's an issue of performance, minimizing the perturbations of monitoring a program's behaviour by the monitoring probes (See Heisenberg Uncertainty Principle).**� We wish to minimize the effect of the Heisenberg uncertainty principle!
- In PartA's main(), print out "Done" after creating the two threads.

- YappyThread should also "spin" after each print.
    - How does a program spin? Add a do-nothing loop that initializes a variable to any number, multiplies it by one and then divides it by one.
    - Note: Using do-nothing loops to spin a program is technique for setting up a concurrency experiment. Spinning is poor technique in actual application programming.
- SleepyThread should sleep() after each print. See the Java API for the *Thread* class for the sleep() method.
- How long should it spin or sleep? You decide. You're going to vary it anyways, below, so make it a variable.

1. Run your program. What got printed out first? Which of the two created threads ran first? What was the pattern of the numbers printed out?
2. Run your program several more times. Is the same thread always run first?
3. Run your program several more times, changing the sleep time and the spin time. What changes do you see?

Part B: No System Services

In this version, the thread classes will now simply "log" its name, instead of printing it to the console. The log is a global variable shared between threads. Logs are used in real-time development when you need to trace the behaviour of your program but you don't want to minimize the impact of tracing due to a comparatively expensive system call to print().

- In PartB's main(), print out the log before terminating. Look in the online API the Java *String* class to find a method that converts a char[] to a *String* for printing.

1. Run your program.
    - Sometimes, students experience an exception when they run their program.� It is usually because they forgot something important about Java arrays. If this happens to you, it�s actually a fantastic opportunity to practice thread debugging.�� Run your program in the debugger and putting a breakpoint on the main()'s print statement. The main() method will stop execution at the breakpoint, but the threads will still execute!
2. Repeat the same experiments as in Part A and look for differences in the behaviours.
3. Now you're going to solve the halting problem using the *join()* method in the *Thread* class. Add the code so that main() joins both threads before printing out the log. Repeat your observations.

Part C: Optional Exploratory Lab on Profiling

You have been exploring thread behavior with a minimum of tool support (Like using Notepad for programming, rather than Eclipse).� I would like to introduce you to a tool called VisualVM.� Unfortunately, it was not incorporated into the lab image[1], so for that reason, I have made this an optional activity. Every challenge is an opportunity though: You will learn how to install plugins into Eclipse.

} It is feasible to do on your lab accounts, but it will be un-done when you log out. You will have to do it each time you log in.

    - If you install this on your own laptop, though, you�d only do it once.
- If it turns out to be manageable, though, we could have some fun with it in later labs and in your project.

Visualvm.java.net:� Home page of the tool
https://visualvm.java.net/eclipse-launcher.html: Eclipse plugin for launching VisualVM
blog.idrsolutions.com/2013/05/setting-up-visualvm-in-under-5-minutes: Installing on Eclipse (and some history)

For those interested in working on Linux (at home): www.baptiste-wicht.com/2010/07/profile-applications-java-visualvm/

Re-run your programs from Part A and B.� Play as you read along with Dr. Dobb: www.drdobbs.com/jvm/visualvm-for-java-development/229403052
- Concentrate on the thread visualization.

Submitting Your Lab Work

- Please submit your two program.
- �Submit your work (even if it's not quite complete) by the end of the lab period in which you are registered.
- Late submissions will **not** be accepted by the instructor or the TAs

---

Updated January 14th, 2014

- [1] Although! �. �*they are included with the latest versions of the Java SE 6 JDK. Look in the bin directory of your local installation to start these tools in standalone mode*.�