SIENAcollege
Computer Science

Computer Science 335
Parallel Processing and High Performance Computing
Siena College
Fall 2021

Topic Notes: Java Threads Basics

While everyone who isn't a C programmer gets up to speed enough for us to start working on parallel programs in C, we're going to use Java for some of our examples.

Some have used Java threads while others have not, so we will review/introduce enough about them to get us going. We start, of course, with a series of multithreaded "Hello, World" programs.

`https://github.com/SienaCSISParallelProcessing/JavaThreadsHello`

First, `HelloThreads.java`. Here, we see the basic mechanism we will use to create threads of execution in Java:

- construct a `Thread` object
- override its `run` method with the code you want the thread to execute, and
- call its `start` method, which creates the new thread of execution within the JVM and operating system, and in turn calls the `Thread` object's `run` method.

When the execution arrives at the `for` loop, the program is like any other standard Java application. The `main` method is executing and nothing else is happening.

However, once we create and start the threads, the program is doing n+1 things. The original `main` method continues to execute, and the `run` method of each thread is executing. Of course in this case, none has a whole lot to do before the program finishes.

If we run it enough, we might see that sometimes, the last printout in `main` occurs before all of the thread printouts have happened.

In some cases, we want to make sure the threads have finished their work (after all, we're soon going to write programs that actually compute something with threads). So we need to make sure the `main` method doesn't proceed until the threads have finished.

The `HelloThreadsWait.java` program does this. Notice that we need to save a reference to each `Thread` object we create, so in addition to calling its `start` method to get it going, we can call its `join` method to wait for it to finish (*i.e.*, for its `run` method to finish executing).

A few things to note here:

- Since we are now using the result of the `Thread` object construction as the right-hand side of an assignment statement, we can no longer call the `start` method right on the created object. The call is moved to a second line.

- It might be tempting to put the `join` calls in the same loop as the constructions and `start` calls, but think about what would happen if we did that?

Finally, we'll often want a program that creates a *team* of n threads to be able to assign each thread a unique sequence number, or thread ID, from o to n-1. These numbers can be useful for deciding which work each thread is supposed to do, among other things.

It might seem like this is easy enough – we have the loop index variable `i` right there. And yes, we will use that loop index variable as the ID. But keep in mind that the value of `i` will be changing as the loop goes around, so each thread will need its own copy of a variable to be able to remember it.

There are many ways to accomplish this. We'll extend the `Thread` class to include an instance variable, where we will store a thread ID as passed to its constructor. So instead of constructing a `Thread` object, we construct a `WorkerThread`, and pass its constructor this additional information. Now, in its `run` method, we can use that variable.

Note that each thread prints out its ID in this example. This demonstrates that even though we create the threads with IDs in increasing order, they will execute in different orders each time! These threads run *asynchronously* and can be run by the JVM in any order. In fact, if we have a multiprocessor, the threads can run truly *concurrently*. Imagine the power this unleashes!