CS 2100: Data Structures & Algorithms 1

Concurrency

Introduction to Threads

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Announcements

- **Final Exam:**
  - Date: Saturday, May 7, 2022
  - Time/Duration: 7:00pm – 9:00pm ET (two hours)
  - Location: TBD (Registrar will confirm rooms, will let you know soon)
- **Make-up Exam:** [Email me if you haven't already]
  - If you have a conflict with the following courses, **email** me:
    - APMA 3100
    - APMA 3140
    - ECON 2020 (sections 001 and 002 only)
  - Make-up Date: Sunday, May 8, 2022
  - *At this time we do not have a time or a location; however, given there are no officially held final exams on this day (May 8) we anticipate the chosen time will suit your schedule*

The Final Exam – Saturday, May 7 (Make-up: May 8)

- Mode: Taken **in-person**
- Duration: **two (2)** hours
- Policies:
  - Closed-book / Closed-notes
  - Closed-Google/Internet (except to access the quiz itself)
  - Closed-Eclipse/other IDE
  - Closed-friend/any other person
  - Closed… everything ☺
  - Can retake **as many quizzes** as you want
  - The work you do must represent your **individual effort**, and involve **no outside assistance** from any one or any resource

- Location of Quizzes: ONLINE AS BEFORE.
  Explicit instructions will be given on the day!
- Students with accommodations with **SDAC**:
  - Please see email that I have sent to you.
  - If you choose to book a testing appointment with SDAC, please do so as soon as possible!
  - You will have your extended time accommodations
- What to bring with you to the final exam:
  - Fully charged **laptop** (+ charging cable)
  - Pen/pencil to write on scratch paper (*not necessary, only if you want*)
  - Student ID card

Introduction to Concurrency / Multithreading

Let’s introduce some basics and some terminology

General Overall Goals

- To understand how multiple **threads** can execute in **parallel**
- To learn to implement threads
- To understand **race conditions** and **deadlocks**
- To avoid corruption of shared objects by using **locks** and **conditions**

- **Content:**
  - *Running Threads*
  - *Terminating Threads*
  - *Race Conditions*
  - *Synchronizing Object Access*
  - *Avoiding Deadlocks*

Motivation

- **Basic idea:** Running code in sequence (i.e., one line of code after another) is fine, and easy. However, what if we could write code that runs in ***parallel*** instead?

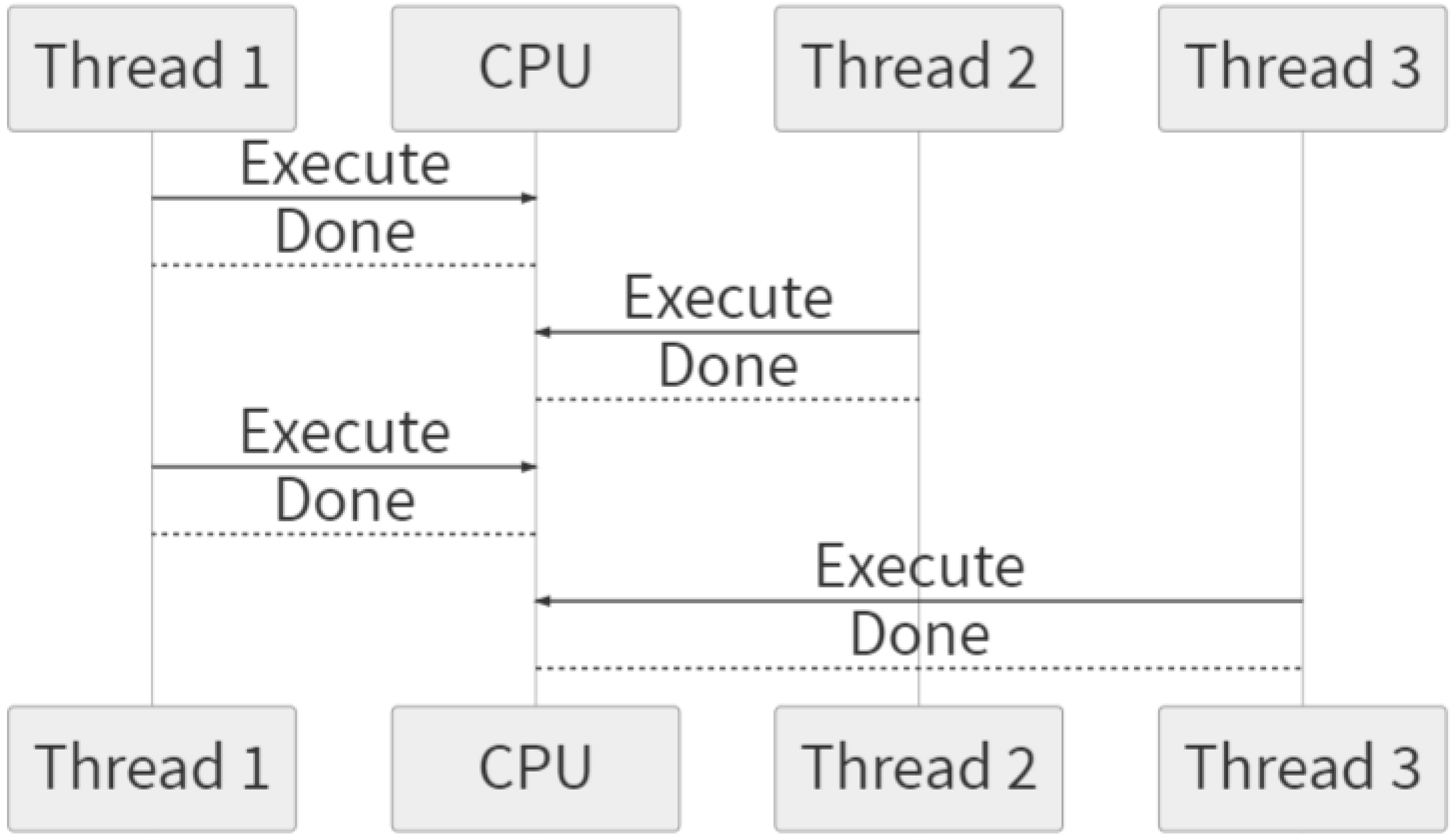
  *Often it is useful for a program to carry out two or more tasks at the same time. This can be achieved effectively by implementing **threads***

- Then, our code would run much ***faster*** right? Running code segment 1 and 2 in parallel is better than executing code 1, then code 2.
  - **Answer**: Well, yes sometimes but not always.

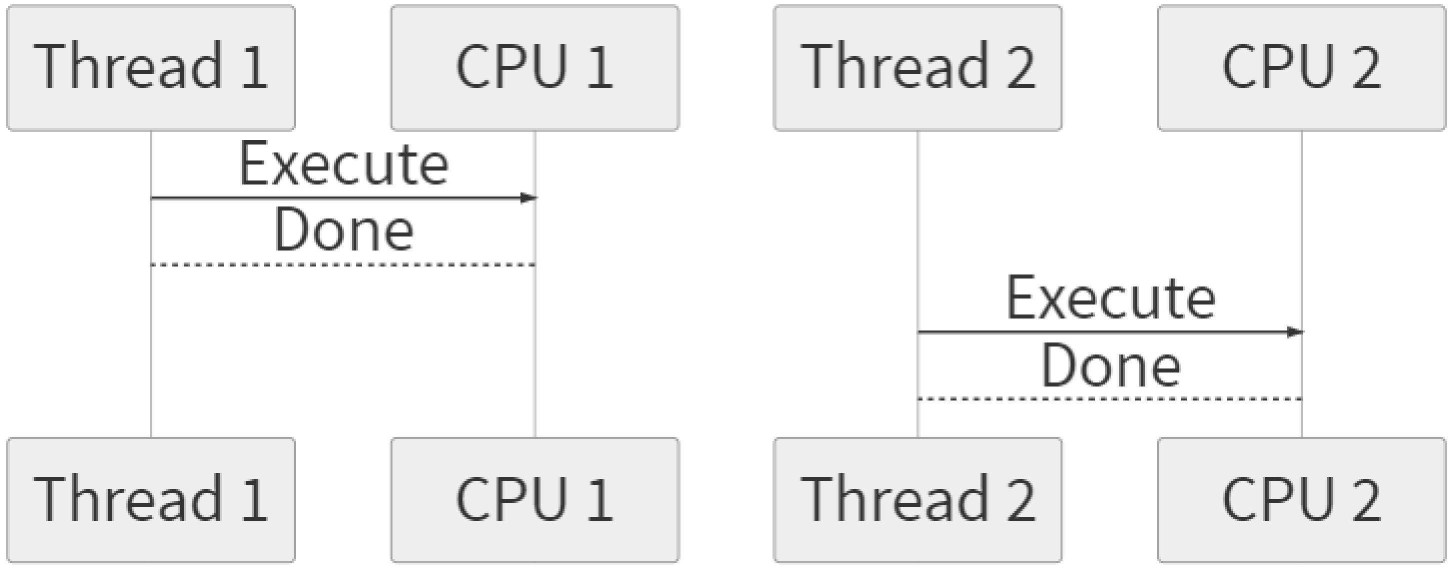
Some Definitions

- **Process:** A program that is running on a machine (e.g., MS Word, Browser, etc.). These processes usually *run in parallel.*
- **Thread:** A *thread* is a piece of **code** that *runs in parallel* within a single process.
    - e.g., Browser may have one thread that handles input from the user and another thread that fetches images to display on the current webpage in parallel.
    - The process has control over all of its threads.
- **CPU (Core):** A CPU is a chip that runs code. If your machine is a **quad-core machine**, then you have four computers in your laptop (good for you!)
- **Resource:** A thing (variable, object, file) that a thread wants to interact with
    - Short version: If threads want to use the same resource, then we have problems.

Single Core Concurrency



Multiple Core Currency



Running Threads

- As mentioned, a thread is a program unit that is executed independently of other parts of the program
- The Java Virtual Machine executes each thread in the program for a ***short amount of time*** [“time slice”]
- This gives the ***impression*** of parallel execution
- If a computer has **multiple central processing units (CPUs)**, then some of the treads can run in parallel, one on each processor

Basic Thread Example

Threads In Java

- Typically, a Java program is a process with **one thread.**
- But, Java provides a nice way to create new threads that run in parallel.
- In comes the Java **Thread** class and **Runnable** interface

- A **Thread Scheduler** runs each **thread** for a short amount of time (time slice)
  - Then the scheduler activates another thread
  - There will always be slight variations in running times – especially when calling operating system services (e.g. input and output)

- There is **no guarantee** about which thread runs first, or **what order threads run in**
  - The “guts” of each thread can be ***interleaved*** like a deck of cards

- **No guarantee** about **where in code a thread is paused** while another takes over.

Making A Thread In Java

1. Create a task to be run in a thread by implementing the **Runnable** **interface**:

```
public interface Runnable
{
   void run(); // one method stub
}
```

2. Place the code for your task into the **run** method of your class (implements Runnable):

```
public class MyRunnable implements Runnable
{  // spawned thread knows to seek run() method
   public void run() // write the body for run() method
   {
       Task statements
       . . .
   }
}
```

Making A Thread In Java

3. Create an object of your subclass: (e.g. “MyRunable”)

```
MyRunnable task = new MyRunnable();
```

4. Construct a **Thread** object from the **MyRunnable** object:

```
Thread t = new Thread(task);
```

5. Call the **start** method (from Thread class) to start the thread: (eventually the **run()** method gets run – scheduled to be invoked)

```
t.start();      // Thread starts and calls task.run()
                // run() method tells the thread what to do
```

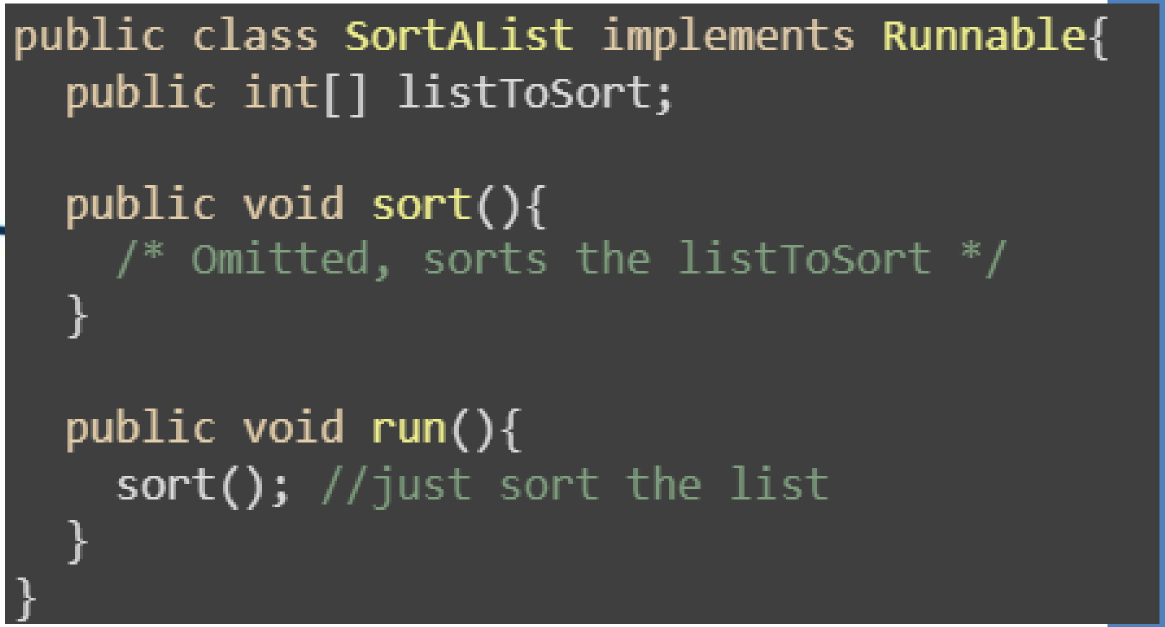
Eclipse DEMO

GreetingRunnable.java – *Basic, one thread example ~ "Hello World!"*

GreetingThreadRunner.java – *Two thread example ~ "Hello" / "Goodbye"*

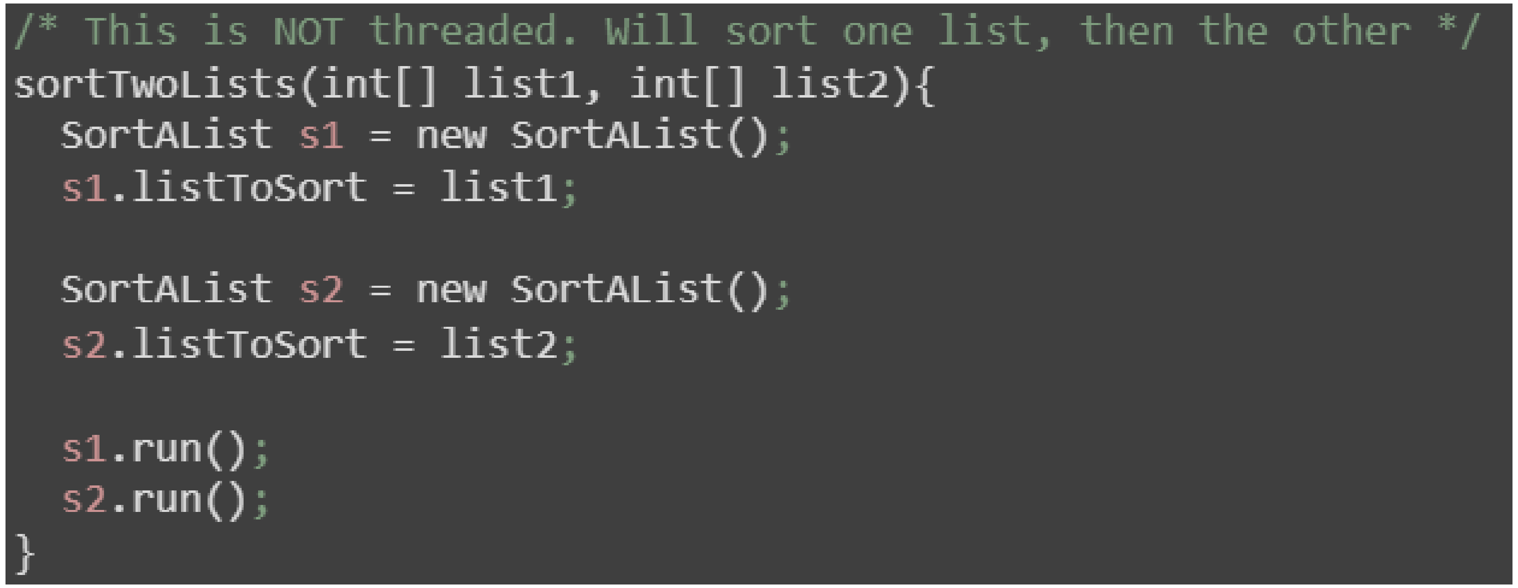
Example: Sorting Two Lists

- Suppose I have two large lists and I need to sort both



```
public class SortAList implements Runnable{
  public int[] listToSort;

  public void sort(){
    /* Omitted, sorts the listToSort */
  }

  public void run(){
    sort(); //just sort the list
  }
}
```

- This example is **NOT threaded**: [sequential]



```
/* This is NOT threaded. Will sort one list, then the other */
sortTwoLists(int[] list1, int[] list2){
  SortAList s1 = new SortAList();
  s1.listToSort = list1;

  SortAList s2 = new SortAList();
  s2.listToSort = list2;

  s1.run();
  s2.run();
}
```

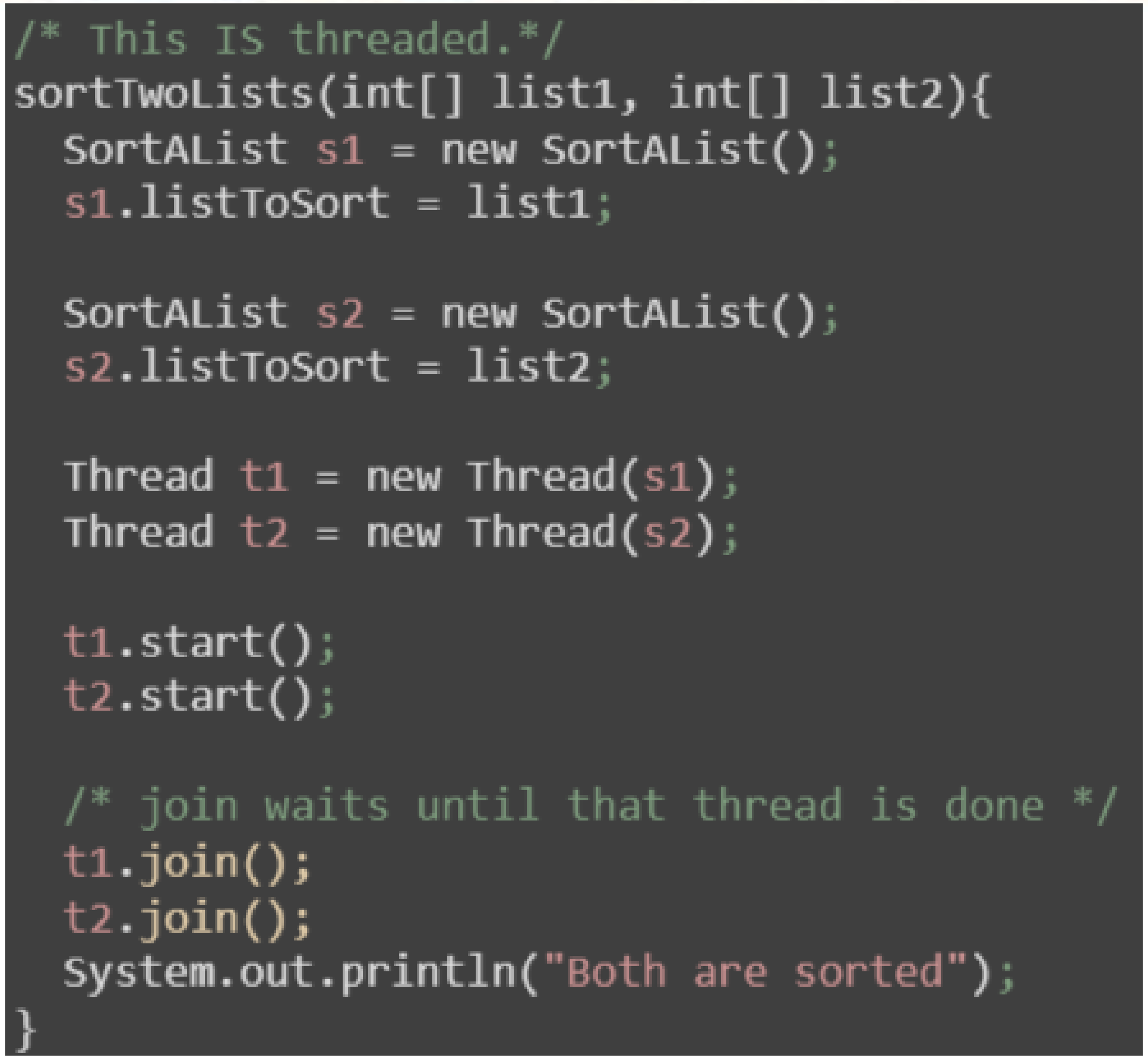
Example: Sorting Two Lists

- This example is **IS threaded**:

- Threads t1 and t2 are spawned
  - Each associated with a list to sort

- The threads are started

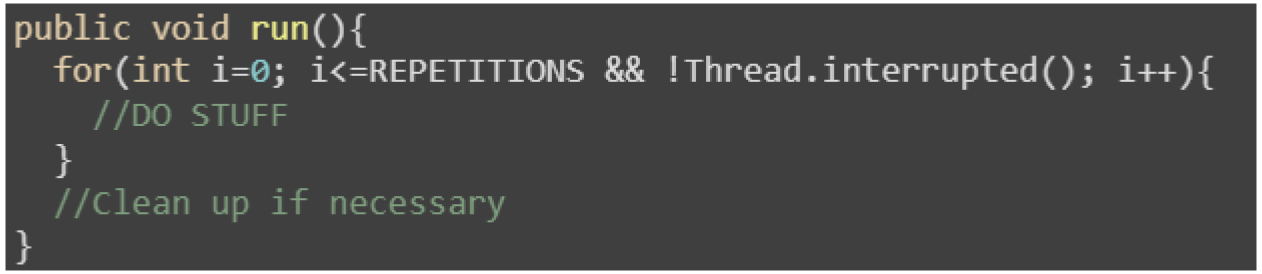
- The lists are sorted “in parallel”



```
/* This IS threaded.*/
sortTwoLists(int[] list1, int[] list2){
  SortAList s1 = new SortAList();
  s1.listToSort = list1;

  SortAList s2 = new SortAList();
  s2.listToSort = list2;

  Thread t1 = new Thread(s1);
  Thread t2 = new Thread(s2);

  t1.start();
  t2.start();

  /* join waits until that thread is done */
  t1.join();
  t2.join();
  System.out.println("Both are sorted");
}
```

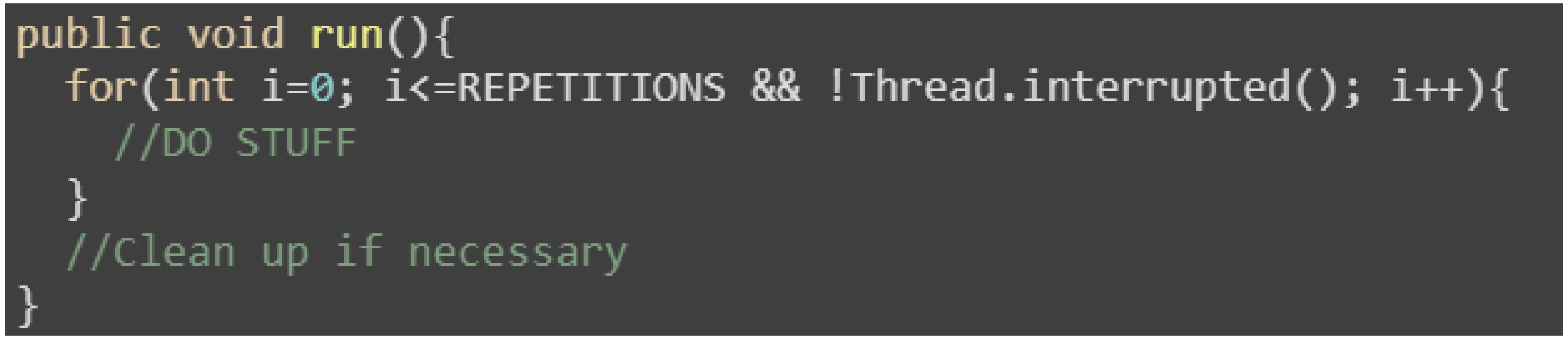
Terminating Threads

- A thread **terminates** when the `run()` method is complete
- Or, you can call:
    - `t.interrupt(); // notifies the tread that it should terminate`
- Does **not** stop the thread (immediately), rather it just sets a **boolean**
    - The run method should check for this interrupt periodically



```
public void run(){
  for(int i=0; i<=REPETITIONS && !Thread.interrupted(); i++){
    //DO STUFF
  }
  //Clean up if necessary
}
```

Terminating Threads



```
public void run(){
  for(int i=0; i<=REPETITIONS && !Thread.interrupted(); i++){
    //DO STUFF
  }
  //Clean up if necessary
}
```

- To suspend execution of a thread, you can call: `sleep()`
  - If a thread is **sleeping** at the time it is interrupted… the thread is **not awake** to check **`Thread.interrupted()`** condition!
  - This is generally NOT a good setup to use

- To better understand how to proceed we have to detour and speak about EXCEPTIONS!