Basic Thread Handling

This lesson shows various thread handling methods with examples.

Joining Threads

In the previous section we discussed how threads can be created. The astute reader would realize that a thread is always created by another thread except for the main application thread. Study the following code snippet. The innerThread is created by the thread which executes the `main` method. You may wonder what happens to the innerThread if the main thread finishes execution before the innerThread is done?

```
class Demonstration {
    public static void main( St

        ExecuteMe executeMe = n
        Thread innerThread = ne
        innerThread.setDaemon(t
        innerThread.start();
    }
}

class ExecuteMe implements Runn

  public void run() {
    while (true) {
      System.out.println("Say H
      try {
        Thread.sleep(500);
      } catch (InterruptedExcep
        // swallow interrupted
      }
    }
  }
}
```

If you execute the above code, you'll see no output. That is because the main thread exits right after starting the innerThread. Once it exits, the JVM also kills the spawned thread. On **line 6** we mark the innerThread thread as a *daemon* thread, which we'll talk about shortly, and is responsible for innerThread being killed as soon as the main thread completes execution. Do bear in mind, that if the main thread context switches just after executing **line** 7, we may see some ouput form the innerThread, till the main thread is context switched back in and exits.

If we want the main thread to wait for the innerThread to finish before proceeding forward, we can direct the main thread to suspend its execution by calling `join` method on the innerThread object right after we `start` the innerThread. The change would look like the following.

```
        Thread innerThread = new Thread(executeMe);
        innerThread.start();
        innerThread.join();
```

If we didn't execute `join` on innerThread and let the main thread continue after innerThread was spawned then the innerThread may get killed by the JVM upon main thread's completion.

Daemon Threads

A daemon thread runs in the background but as soon as the main application thread exits, all daemon threads are killed by the JVM. A thread can be marked daemon as follows:

```
innerThread.setDaemon(true);
```

Note that in case a spawned thread isn't marked as daemon then even if the main thread finishes execution, JVM will wait for the spawned thread to finish before tearing down the process.

Sleeping Threads

A thread can be made dormant for a specified period using the `sleep` method. However, be wary to not use sleep as a means for coordination among threads. It is a common newbie mistake. Java language framework offers other constructs for thread synchronization that'll be discussed later.

```
class SleepThreadExample {
    public static void main( String args[] ) throws Exception {
        ExecuteMe executeMe = new ExecuteMe();
        Thread innerThread = new Thread(executeMe);
        innerThread.start();
        innerThread.join();
        System.out.println("Main thread exiting.");
    }
    static class ExecuteMe implements Runnable {

        public void run() {
            System.out.println("Hello. innerThread going to sleep");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                // swallow interrupted exception
            }
        }
    }
}
```

In the above example, the innerThread is made to sleep for 1 second and from the output of the program, one can see that main thread exits only after innerThread is done processing. If we remove the `join` statement on ***line-6***, then the main thread may print its statement before innerThread is done executing.

Interrupting Threads

In the previous code snippets, we wrapped the calls to `join` and `sleep` in try/catch blocks. Imagine a situation where if a rogue thread sleeps forever or goes into an infinite loop, it can prevent the spawning thread from moving ahead because of the `join` call. Java allows us to force such a misbehaved thread to come to its senses by interrupting it. An example appears below.

```
class HelloWorld {
    public static void main( String args[] ) throws InterruptedException {
        ExecuteMe executeMe = new ExecuteMe();
        Thread innerThread = new Thread(executeMe);
        innerThread.start();

        // Interrupt innerThread after waiting for 5 seconds
        System.out.println("Main thread sleeping at " + +System.currentTimeMillis() / 1000);
        Thread.sleep(5000);
        innerThread.interrupt();
        System.out.println("Main thread exiting at " + +System.currentTimeMillis() / 1000);
    }

    static class ExecuteMe implements Runnable {

        public void run() {
            try {
                // sleep for a thousand minutes
                System.out.println("innerThread goes to sleep at " + System.currentTimeMillis
                Thread.sleep(1000 * 1000);
            } catch (InterruptedException ie) {
                System.out.println("innerThread interrupted at " + +System.currentTimeMillis(
            }
        }
    }

}
```