CS 2340 Objects and Design

Java Threads

Christopher Simpkins
chris.simpkins@gatech.edu

Processes and Threads

A *process* is a unit of execution within the operating system. Each process has its own

- stack,
- program counter, and
- memory space (heap).

A *thread* is a unit of execution that belongs to a process. Typically, threads

- have their own stack and program counter, and
- **share** memory space with the other threads of a process.

Each process has at least one thread.

Creating Thread Objects (Runnables)

Two ways to create a thread:

- Subclass `Thread`
- Implement the `Runnable` interface

Better to use `Runnable`, which is more flexible.

```
public interface Runnable {
    /**
     * When an object implementing interface Runnable is used to
     * create a thread, starting the thread causes the object's
     * run method to be called in that separately executing thread.
     */
    void run();
}
```

Any class can implement the `Runnable` interface and be used to create threads.

Implementing Runnable

Here's a simple Runnable, MeowTask.java:

```
public class MeowTask implements Runnable {

    public void run() {
        for (int i = 0; i < 9; i++) {
            System.out.println(Thread.currentThread().getName()
                +": Meow ...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() +
                    ":Don't stop me meow, I have " + (9 - i)
                    + " to go!");
                // What if we don't return?
                return;
            }
        }
        System.out.println("Meow!");
    }
}
```

Creating and Running a Thread

And here's a simple program that uses it to create and run a thread (BasicMeow.java):

```
public class BasicMeow {
    public static void main(String[] args) throws InterruptedException{
        Thread t = new Thread(new MeowTask(), "Foster");
        t.start();
        // What if we don't join?
        t.join();
        System.out.println("Done.");
    }
}
```

- `Thread` constructor takes a `Runnable` and optionally a name for the thread (a good practice to aid in debugging)
- Calling `start` maked the thread eligible for execution by the OS's scheduler
- This application has two threads: the `Main` thread and the `Foster` thread

`join()`ing a Thread

Notice that in BasicMeow.java we call `join` on the thread after we start it.

```
public class BasicMeow {

    public static void main(String[] args) throws InterruptedException
    {
        Thread t = new Thread(new MeowTask(), "Foster");
        t.start();
        // What if we don't join?
        t.join();
        System.out.println("Done.");
    }
}
```

Calling `join()` on a thread causes a thread to wait for the other thread to terminate. Let's run BasicMeow.java to see what happens if we don't join the `Foster` thread ...

Sleeping and Interrupting

Notice that in MeowTask.java's `run` method we sleep:

```
public void run() {
    for (int i = 0; i < 9; i++) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { }
    }
}
```

Calling `Thread.sleep()`

- makes processor time available to other threads,
- paces execution (intentional pauses), and
- supports interruption

Interrupting

An interrupt signals a thread to suspend execution and do something, typically terminate. The most common way to support interruption is to call methods that throw `InterruptedExceptoin`, like `Thread.sleep()` and `Object.wait()` (more later).
In MeowTask.java's `run` method we return from the `run` method when we get an `InterruptedException`:

```
public void run() {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // What if we don't return?
                return;
            }
        }
```

If we didn't `return` in the catch block the thread would keep running until the program terminated. Let's look at TimeoutMeow.java and see this in action ...