Session Overview

- Threads Intro
- Extending `Thread`
- Implementing `Runnable`
- Implementing `Runnable` with an Inner Class
- Self-Running Objects
- Methods on `Thread`
- Determining the Current Thread
- Putting the Current Thread to Sleep
- Correcting for Inaccurate Sleep
- Interrupting
- Thread Priorities
- Lab – CountdownTimer

Threads and Java

- Threads:
    - allow an application to do more that one task simultaneously. Examples:
        - a word processor printing in the background
        - a long-running search that allows you to interrupt the search or to do other things while the search is running.
        - an email program that checks for mail in the background.
        - a web browser that retrieves multiple images simultaneously.
    - applications that do this are called "**multithreaded**".
    - classes that are written in a manner that safely support simultaneous access by multiple threads are said to be "**thread-safe**" or "**multithread-safe**"
    - Java has built-in, intimate support for multithreaded programming.

- A thread is a lightweight path of execution though an application. A process would be a considered heavyweight, but a thread runs within a process and is considered "lighter". A process is a program in motion (currently running).

- Core threading support classes and interfaces (like `Thread` and `Runnable`) are in the `java.lang` package.

Extending the `Thread` Class

- The simplest way to spawn a new thread is to subclass the `Thread` class.
- This is generally not a good technique, but is a simple one that's good to start with.

ExtendThread.java:

```
public class ExtendThread extends Thread {
    public void run() {
        System.out.println("Inside run() method");
    }
}
```

ExtendThreadDemo.java:

```
public class ExtendThreadDemo extends Object {
    public static void main(String[] args) {
        System.out.println("Inside main() - about to construct");
        ExtendThread t = new ExtendThread();

        System.out.println("Inside main() - about to start()");
        t.start();

        System.out.println("Inside main() - back from start()");
    }
}
```

Possible Output:

```
Inside main() - about to construct
Inside main() - about to start()
Inside main() - back from start()
Inside run() method
```

--when `start()` is called, a request is put in to the VM to spawn a new thread (sometime soon). The thread calling `start()` returns right away and continues execution with whatever code follows `start()`. Meanwhile, the VM spawns a new thread and has this new thread call the `run()` method. At this point, there are two threads running simultaneously. Note that `run()` is ***not*** called directly.

- How can two threads run simultaneously if there is only one processor?
  - simultaneous execution is simulated on single processor machines by letting each thread run for a fraction of a second.
  - when the processor switches between threads, it's called a context switch.

Implementing the `Runnable` Interface

- For the vast majority of cases, a better alternative to extending `Thread` is ***implementing*** the `Runnable` interface:

```
public interface Runnable {
    void run();
}
```

  - Why not subclass `Thread`?
    - If we don't need a special ***kind*** of `Thread`, then why subclass? We only need to use a `Thread`, not a specialization of it.
    - If we extend `Thread`, then anyone can call any of the `public` methods on `Thread`—maybe some methods that we don't want them to call like `suspend()`, `stop()`, `setPriority()`, `start()`, or whatever!
    - The functionality that we have may need to be run again and again. An instance of `Thread` can only be run (started) once.

- An instance of a class that has implemented the `Runnable` interface can be passed to one of the constructors of `Thread`:

```
public Thread(Runnable r)
```

ImplementRunnable.java:

```
public class ImplementRunnable extends Object implements Runnable {
    public void run() {
        System.out.println("Inside run() method");
    }
}
```

ImplementRunnableDemo.java:

```
public class ImplementRunnableDemo extends Object {
    public static void main(String[] args) {
        System.out.println("Inside main() - about to construct");
        Runnable r = new ImplementRunnable();

        System.out.println("Inside main() - creating/starting Thread");
        Thread t = new Thread(r);
        t.start();

        System.out.println("Inside main() - back from start()");
    }
}
```

Possible Output:

```
Inside main() - about to construct
Inside main() - creating/starting Thread
Inside main() - back from start()
Inside run() method
```

Implementing `Runnable` with an Inner Class

- Although implementing `Runnable` exposes less than extending `Thread`, we still have a `public run()` method hanging out there for anyone to call.
- We usually don't want to burden a user of our class with the task of starting a thread for us and/or knowing that the class is a `Runnable`.
- We can use an anonymous inner class that implements `Runnable` on-the-fly and have it call a `private` method on the enclosing class. This effectively hides the `public` method `run()` from the outside:

InnerRun.java:

```
public class InnerRun extends Object {
    //...
    public void start() {
        Runnable r = new Runnable() {
                public void run() {
                    runWork();
                }
            };

        Thread t = new Thread(r);
        t.start();
    }

    private void runWork() {
        System.out.println("Inside the private runWork() method");
    }
}
```

InnerRunDemo.java:

```
public class InnerRunDemo extends Object {
    public static void main(String[] args) {
        System.out.println("Inside main() - about to construct");
        InnerRun ir = new InnerRun();

        System.out.println("Inside main() - calling start()");
        ir.start();

        System.out.println("Inside main() - back from start()");
    }
}
```

Possible Output:

```
Inside main() - about to construct
Inside main() - calling start()
Inside main() - back from start()
Inside the private runWork() method
```

Self-Running Objects

- Self-running objects automatically spawn an internal thread during construction eliminating the need for "outside" code to know that threads are being used.
- See Chapter 11 of "Java Thread Programming" for details on this pattern.
- Example: **`SelfRunPattern.java`**
- Example: **`SelfRunMore.java`**

`SelfRunPattern.java`

```
 1: public class SelfRunPattern extends Object {
 2:     public SelfRunPattern() {
 3:         // ...
 4:         // do the regular constructor stuff
 5:         // ...
 6:
 7:         // Just before the constructor returns,
 8:         // spawn a new thread to call runWork()
 9:         Runnable r = new Runnable() {
10:                 public void run() {
11:                     runWork();
12:                 }
13:             };
14:
15:         Thread t = new Thread(r);
16:         t.start();
17:     }
18:
19:     private void runWork() {
20:         // gets called by the internal thread
21:         // ...
22:     }
23: }
```

SelfRunMore.java

```
 1: public class SelfRunMore extends Object {
 2:     // See Chapter 11 - "Self-Running Objects"
 3:     // of "Java Thread Programming" for more.
 4:
 5:     private Thread internalThread;
 6:     private volatile boolean noStopRequested;
 7:
 8:     public SelfRunMore() {
 9:         // ...
10:         // do the regular constructor stuff
11:         // ...
12:
13:         // Just before the constructor returns,
14:         // spawn a new thread to call runWork()
15:
16:         noStopRequested = true;
17:         Runnable r = new Runnable() {
18:                 public void run() {
19:                     runWork();
20:                 }
21:             };
22:
23:         // Give the internal thread a name
24:         internalThread = new Thread(r, "SelfRunMore-internal");
25:         internalThread.start();
26:     }
27:
28:     private void runWork() {
29:         // gets called by the internal thread
30:         // ...
31:
32:         while ( noStopRequested ) {
33:             // ...
34:         }
35:     }
36:
37:     public void stopRequest() {
38:         noStopRequested = false;
39:         internalThread.interrupt();
40:     }
41:
42:     public boolean isAlive() {
43:         return internalThread.isAlive();
44:     }
45: }
```

Some of the methods on `Thread`:

- `start()` –asynchronously requests that the VM spawn a new thread.

- `run()` –is called by the newly spawned thread "sometime after `start()` is called". It is possible that the new thread will be inside `run()` before the original thread returns from `start()`. It is also possible that the original thread will have returned from `start()` before the new thread calls `run()`. Neither behavior should be counted on, this—like most thread stuff—is non-deterministic.

- `stop()` –causes the target thread to cease execution.
  - method has been deprecated as of 1.2.
  - it was always dangerous and is not recommended as a way of getting a thread to die.
  - it's prone to leaving objects in an inconsistent state by suddenly releasing any locks that might be held (locks acquired via synchronized).

- `suspend()` –causes the target thread to pause execution.
  - method has been deprecated as of 1.2.
  - it is prone to deadlocks as is keeps holding any locks it has while suspended.

- `resume()` –causes the target thread to continue execution after being paused by `suspend()`.
  - method has been deprecated as of 1.2—not because it's dangerous, but because it's no longer needed without `suspend()`.
  - See the example starting on ***page 91 of "Java Thread Programming"*** for a good replacement for `stop()`, `supend()`, and `resume()`.

- `interrupt()` –signals the target thread that it should take notice—usually used to signal that a thread should clean up and die gracefully.

- `isAlive()` –checks the current state of a thread. A thread is considered to be alive from just before `run()` is called until just after `run()` returns.

- `getName()` –returns the name if the thread (does not have to be unique).

- `setName()` –changes the name of the thread. Thread names can also be specified by passing a string to some of the constructors of `Thread`.

Determining the Current Thread

- **`Thread.currentThread()`** –used to determine which thread is calling the **`currentThread()`** method.

    - a useful diagnostic and learning technique is to create a method to help print messages:

```
public static void print(String msg) {
    String name = Thread.currentThread().getName();
    System.out.println(name + ": " + msg);
}
```

- Example: **`NameCheck.java`**

`NameCheck.java`

```
 1: public class NameCheck extends Object {
 2:     public NameCheck(String name) {
 3:         Runnable r = new Runnable() {
 4:                 public void run() {
 5:                     runWork();
 6:                 }
 7:             };
 8:
 9:         Thread t = new Thread(r, name);
10:         t.start();
11:     }
12:
13:     private void runWork() {
14:         try {
15:             String threadName = Thread.currentThread().getName();
16:
17:             for ( int i = 0; i < 10; i++ ) {
18:                 System.out.println("threadName=" + threadName + ", i=" + i);
19:                 Thread.sleep(1000);
20:             }
21:         } catch ( InterruptedException x ) {
22:             // ignore and return from runWork();
23:         }
24:     }
25:
26:     public static void main(String[] args) {
27:         new NameCheck("apple");
28:         new NameCheck("banana");
29:         new NameCheck("orange");
30:         new NameCheck("pear");
31:         new NameCheck("pineapple");
32:
33:         String name = Thread.currentThread().getName();
34:         System.out.println("name of thread running main() is '" + name + "'");
35:     }
36: }
```

Possible Output

```
 1: threadName=apple, i=0
 2: threadName=banana, i=0
 3: threadName=orange, i=0
 4: threadName=pear, i=0
 5: name of thread running main() is 'main'
 6: threadName=pineapple, i=0
 7: threadName=apple, i=1
 8: threadName=banana, i=1
 9: threadName=orange, i=1
10: threadName=pear, i=1
11: threadName=pineapple, i=1
12: threadName=apple, i=2
13: threadName=banana, i=2
14: threadName=pear, i=2
15: threadName=pineapple, i=2
16: threadName=orange, i=2
17: threadName=apple, i=3
18: threadName=banana, i=3
19: threadName=pear, i=3
20: threadName=pineapple, i=3
21: threadName=orange, i=3
22: threadName=apple, i=4
23: threadName=banana, i=4
24: threadName=orange, i=4
25: threadName=pear, i=4
26: threadName=pineapple, i=4
27: threadName=apple, i=5
28: threadName=banana, i=5
29: threadName=orange, i=5
30: threadName=pear, i=5
31: threadName=pineapple, i=5
32: threadName=apple, i=6
33: threadName=banana, i=6
34: threadName=orange, i=6
35: threadName=pear, i=6
36: threadName=pineapple, i=6
37: threadName=apple, i=7
38: threadName=banana, i=7
39: threadName=orange, i=7
40: threadName=pear, i=7
41: threadName=pineapple, i=7
42: threadName=apple, i=8
43: threadName=banana, i=8
44: threadName=orange, i=8
45: threadName=pear, i=8
46: threadName=pineapple, i=8
47: threadName=apple, i=9
48: threadName=banana, i=9
49: threadName=orange, i=9
50: threadName=pear, i=9
51: threadName=pineapple, i=9
```

Putting the Current Thread to Sleep

- **`Thread.sleep()`** –used to put the current thread to sleep for approximately the specified number of milliseconds.
  - a thread can only put itself to sleep.
  - the number of milliseconds (1/1000th of a second) to sleep is specified.
  - if the sleeping thread is interrupted by another thread, it throws an **`InterruptedException`**.
  - to put the current thread to sleep for 3 seconds:

```
try {
    Thread.sleep(3000);
} catch ( InterruptedException x ) {
    x.printStackTrace();
}
```

  - because this sleep time is approximate, we need to not depend upon it's accuracy.

Correcting for Inaccurate Sleep

- Thread.sleep() is not 100% accurate—we might return a few milliseconds too soon or too late.
- We need to keep from drifting too slow or too fast if we sleep over and over in a loop
- We do this by adjusting our sleep time each pass through the loop.
- Example: SecondCounter (from book)

Interrupting Threads

- One thread can interrupt another thread.
- Used as a signal—typically that the interrupted thread should gracefully cleanup and die.
- The target thread is interrupted when another thread invokes:
  ```
  public void interrupt()
  ```
  on the target.

- Calls to `interrupt()` return immediately—typically before the interrupted thread has noticed the interruption.
- Threads can be interrupted at any time (not just when sleeping).

- If threadA wants to interrupt threadB, this is done:

```
// threadA is running this code...
threadB.interrupt();
// threadA continues to run...
```

- See Chapter 5 of Java Thread Programming for more on interrupts.
- Example: `InterruptDemo.java`

InterruptDemo.java

```
 1: public class InterruptDemo extends Object {
 2:     private Thread internalThread;
 3:
 4:     public InterruptDemo() {
 5:         Runnable r = new Runnable() {
 6:                 public void run() {
 7:                     runWork();
 8:                 }
 9:             };
10:
11:         internalThread = new Thread(r);
12:         internalThread.start();
13:     }
14:
15:     private void runWork() {
16:         long startTime = System.currentTimeMillis();
17:
18:         try {
19:             print("in runWork() - about to sleep");
20:             Thread.sleep(5000);
21:             print("in runWork() - slept for 5 sec.");
22:         } catch ( InterruptedException x ) {
23:             double elapsedSec = (System.currentTimeMillis() -
24:                 startTime) / 1000.0;
25:             print("interrupted after only " +
26:                 elapsedSec + " sec.");
27:         }
28:     }
29:
30:     public void wakeUp() {
31:         internalThread.interrupt();
32:     }
33:
34:     public static void print(String msg) {
35:         String threadName = Thread.currentThread().getName();
36:         System.out.println(threadName + ": " + msg);
37:     }
38:
39:     public static void main(String[] args) {
40:         InterruptDemo id = new InterruptDemo();
41:         print("about to sleep for 2 sec.");
42:
43:         try {
44:             Thread.sleep(2000);
45:         } catch ( InterruptedException x ) {
46:             // ignore
47:         }
48:
49:         print("about to invoke wakeUp()");
50:         id.wakeUp();
51:         print("back from wakeUp()");
52:     }
53: }
```

Output

```
1: main: about to sleep for 2 sec.
2: Thread-1: in runWork() - about to sleep
3: main: about to invoke wakeUp()
4: Thread-1: interrupted after only 2.013 sec.
5: main: back from wakeUp()
```

Thread Prioritization

- The `Thread` API allows for the setting and getting of a thread's suggested priority:

```
public void setPriority(int newPriority)
public int getPriority()
```

- Higher priority threads generally get to run more often on the processor than lower priority threads—but this is not guaranteed.
- On some implementations, lower-priority threads don't get to run ***at all*** until all of the threads of a higher priority are blocked (sleeping, waiting on I/O, etc.). You should be sure that higher priority threads occasionally block.

- A thread can voluntarily relinquish its turn on the processor by invoking:

```
Thread.yield();
```

- Be careful not to call yield too often because of the overhead of a context switch. An occasional call to `Thread.yield()` during a CPU-intensive section of code can sometimes be helpful in thread scheduling.