Introduction to Java™

Module 9: Threads

Introduction to Threads

- Objects provide a way to divide a program up into independent sections.
- Often, you also need to turn a program into separate, independently-running subtasks.
- Each of these independent subtasks is called a *thread.*
- You program as if each thread runs by itself and has the CPU to itself.

Introduction to Threads

- A *process* is a self-contained running program with its own address space.
- A *multitasking* operating system is capable of running more than one process (program) at a time.
- A thread is a single sequential flow of control within a process.
- A single process can thus have multiple concurrently executing threads.

Introduction to Threads

- There are many possible uses for multithreading
  - In general, you'll have some part of your program tied to a particular event or resource
    (and you don't want to hang up the rest of your program because of that).
  - So you create a thread associated with that event or resource and let it run independently of the main program.
  - A good example is a "quit" button
    you don't want to be forced to poll the quit button in every piece of code you write in your program and yet you want the quit button to be responsive, as if you *were* checking it regularly.
  - In fact, one of the most immediately compelling reasons for multithreading is to produce a responsive user interface.

Inheriting From Thread

- The simplest way to create a thread is to inherit from class **Thread**, which has all the wiring necessary to create and run threads.
- The most important method for **Thread** is **run( )**, which you must override to make the thread do your bidding.
- Thus, **run( )** is the code that will be executed "simultaneously" with the other threads in a program.

Thread Example

```
public class SimpleThread extends Thread {
     private int countDown = 5;
     private int threadNumber;
     private static int threadCount = 0;
     public SimpleThread() {
       threadNumber = ++threadCount;
       System.out.println("Making " + threadNumber);
     }
     public void run() {
       while(true) {
         System.out.println("Thread " +
           threadNumber + "(" + countDown + ")");
         if(--countDown == 0) return;
       }
     }
     public static void main(String[] args) {
       for(int i = 0; i < 5; i++)
         new SimpleThread().start();
       System.out.println("All Threads Started");
     }
   }
```

Your First Thread

- A **run( )** method virtually always has some kind of loop that continues until the thread is no longer necessary.
- In the previous example, in **main( )** you can see a number of threads being created and run.
- The special method that comes with the **Thread** class is **start( )** (performs special initialization for the thread and then calls **run( )**).
- So the steps are:
  - the constructor is called to build the object.
  - **start( )** configures the thread and calls **run( )**.
- If you don't call **start( )** the thread will never be started.

Runnable Interface

- Java does not support multiple inheritance
  - Instead, use interfaces
  - Until now, inherited from class Thread, overrode run
- Multithreading for an already derived class
  - Implement interface Runnable (java.lang)
    - ✓ New class objects "are" Runnable objects
  - Override run method
    - ✓ Controls thread, just as deriving from Thread class
    - ✓ In fact, class Thread implements interface Runnable
- Create new threads using Thread constructors
  - Thread(runnableObject)
  - Thread(runnableObject, threadName)

Daemon Threads

- A "daemon" thread is one that is supposed to provide a general service in the background as long as the program is running.
- Not part of the essence of the program. Thus, when all of the **non-daemon** threads complete the program is terminated.
- You can find out if a thread is a daemon by calling **isDaemon( )**, and you can turn the daemonhood of a thread on and off with **setDaemon( )**.
- If a thread is a daemon, then any threads it creates will automatically be daemons.

Example Daemon Threads

```
class Daemon extends Thread {
     private static final int SIZE = 10;
     private Thread[] t = new Thread[SIZE];
     public Daemon() {
       setDaemon(true);
       start();
     }
     public void run() {
       for(int i = 0; i < SIZE; i++)
         t[i] = new DaemonSpawn(i);
       for(int i = 0; i < SIZE; i++)
        System.out.println(
             "t[" + i + "].isDaemon() = "
             + t[i].isDaemon());
       while(true)
         yield();
     }
   }
```

Example Daemon Threads

```
class DaemonSpawn extends Thread {
  public DaemonSpawn(int i) {
    System.out.println("DaemonSpawn "+i+" started");
    start();
  }
  public void run() {
    while(true)
      yield();
  }
}
```

Example Daemon Threads

```
public class Daemons {
  public static void main(String[] args) {
    Thread d = new Daemon();
    System.out.println("d.isDaemon() = " + d.isDaemon());
    // Allow the daemon threads to finish
    // their startup processes:
    BufferedReader stdin =
      new BufferedReader(new InputStreamReader(System.in));
    System.out.println("Waiting for CR");
    try {
      stdin.readLine();
    } catch(IOException e) {}
  }
}
```

Example Daemon Threads

- The **Daemon** thread sets its daemon flag to "true" and then spawns a bunch of other threads to show that they are also daemons.
- Then it goes into an infinite loop that calls **yield( )** to give up control to the other processes.
- There's nothing to keep the program from terminating once **main( )** finishes its job since there are nothing but daemon threads running.
- So that you can see the results of starting all the daemon threads, **System.in** is set up to read so the program waits for a carriage return before terminating.

Multithreading Issues

- Sharing limited resources:
  - You can think of a single-threaded program as one lonely entity moving around through your problem space and doing one thing at a time.
  - You never have to think about the problem of two entities trying to use the same resource at the same time.
  - With multithreading you now have the possibility of two or more threads trying to use the same limited resource at once.
  - Colliding over a resource must be prevented or else you'll have two threads trying to access the same resource at the same time
    (e.g. print to the same printer, or adjust the same value, etc.)

Multithreading Issues

- A fundamental problem with using threads:

  You never know when a thread might be run.
- Sometimes you don't care if a resource is being accessed at the same time you're using it.
- Need some way to prevent two threads from accessing the same resource. (at least during critical periods)
- Preventing this kind of collision is simply a matter of putting a lock on a resource when one thread is using it.
- The first thread that accesses a resource locks it, and then the other threads cannot access that resource until it is unlocked.

How Java Shares Resources

- Java has built-in support to prevent collisions over one kind of resource:
  - The memory in an object.
  - Since you typically make the data elements of a class **private** and access that memory only through methods, you can prevent collisions by making a particular method **synchronized**.
  - Only one thread at a time can call a **synchronized** method for a particular object.
  - Here are simple **synchronized** methods:

```
synchronized void f() { /* ... */ }
synchronized void g(){ /* ... */ }
```

How Java Shares Resources

- Each object contains a single lock (also called a *monitor*) that is automatically part of the object (you don't have to write any special code).
- When you call any **synchronized** method, that object is locked and no other **synchronized** method of that object can be called until the first one finishes and releases the lock.
- In the example above, if **f( )** is called for an object, **g( )** cannot be called for the same object until **f( )** is completed.

How Java Shares Resources

- There's a single lock that's shared by all the **synchronized** methods of a particular object.
- There's also a single lock per class (as part of the **Class** object for the class), so that **synchronized static** methods can lock each other out from **static** data on a class-wide basis.
- NOTE:
  If you want to guard some other resource from simultaneous access by multiple threads, you can do so by forcing access to that resource through **synchronized** methods.

More on synchronized

- We can remove the **synchronized** keyword from the entire method and instead put a **synchronized** block around the critical lines.
- But what object should be used as the lock? The current object (**this**)!
- Of course, all synchronization depends on programmer diligence: every piece of code that can access a shared resource must be wrapped in an appropriate synchronized block.

Example

```
public void run() {
    while (true) {
      synchronized(this) {
        t1.setText(Integer.toString(count1++));
        t2.setText(Integer.toString(count2++));
      }
      try {
        sleep(500);
      } catch (InterruptedException e){}
    }
  }
```

Thread States

- A thread can be in any one of four states:
  - *New*: the thread object has been created but it hasn't been started yet so it cannot run.
  - *Runnable*: This means that a thread *can* be run when the time-slicing mechanism has CPU cycles available for the thread.
  - *Dead*: the normal way for a thread to die is by returning from its **run( )** method. You can also call **stop( )**.
  - *Blocked*: the thread could be run but there's something that prevents it. Until a thread re-enters the runnable state it won't perform any operations.

Becoming Blocked

- A thread can become blocked for five reasons:
  - You've put the thread to sleep by calling **sleep(milliseconds)**.
  - You've suspended the execution of the thread with **suspend( )**. It will not become runnable again until the thread gets the **resume( )** message.
  - You've suspended the execution of the thread with **wait( )**. It will not become runnable again until the thread gets the **notify( )** or **notifyAll( )** message.
  - The thread is waiting for some IO to complete.
  - The thread is trying to call a **synchronized** method on another object and that object's lock is not available.

More on Blocking

- You can also call **yield( )** (a method of the **Thread** class) to voluntarily give up the CPU so that other threads can run.
- However, the same thing happens if the scheduler decides that your thread has had enough time and jumps to another thread.
- That is, nothing prevents the scheduler from re-starting your thread.
- When a thread is blocked, there's some reason that it cannot continue running.

Wait and Notify

- The point with both sleep( ) and suspend( ) is that *do not* release the lock as they are called.
- On the other hand, the method wait( ) *does* release the lock when it is called, which means that other synchronized methods in the thread object could be called during a wait( ).
- You'll also see that there are two forms of **wait( )**.
  - The first takes an argument in milliseconds that has the same meaning as in **sleep( )**
  - The second form takes no arguments

Wait and Notify

- You can put a **wait( )** inside any **synchronized** method, regardless of whether there's any threading going on inside that particular class.
- In fact, the *only* place you can call **wait( )** is within a **synchronized** method or block.
- If you call **wait( )** or **notify( )** within a method that's not **synchronized**, the program will compile, but when you run it you'll get an **IllegalMonitorStateException**
  with the somewhat non-intuitive message "current thread not owner."

Wait and Notify

- You can call **wait( )** or **notify( )** only for your own lock.
  Again, you can compile code that tries to use the wrong lock, but it will produce the same **IllegalMonitorStateException** message as before.
- You can't fool with someone else's lock, but you can ask another object to perform an operation that manipulates its own lock.
- So one approach is to create a **synchronized** method that calls **notify( )** for its own object.

Deadlock

- Because threads can become blocked *and* because objects can have **synchronized** methods that prevent threads from accessing that object until the synchronization lock is released, it's possible for one thread to get stuck waiting for another thread, which in turn waits for another thread, etc., until the chain leads back to a thread waiting on the first one.
- This is called *deadlock*. The claim is that it doesn't happen that often, but when it happens to you it's frustrating to debug.
- There is no language support to help prevent deadlock.

Too Many Threads

- You must watch to see that you don't have "too many threads".
- If you do, you must try to use techniques to "balance" the number of threads in your program.
- If you see performance problems in a multithreaded program you now have a number of issues to examine:
  - Do you have enough calls to **sleep( ), yield( ),** and/or **wait( )**?
  - Are calls to **sleep( )** long enough?
  - Are you running too many threads?
  - Have you tried different platforms and JVMs?
- Issues like this are one reason that multithreaded programming is often considered an art.

Summary

- It is vital to learn when to use multithreading and when to avoid it.
- The main drawbacks to multithreading are:
  - Slowdown while waiting for shared resources
  - Additional CPU overhead required to manage threads
  - Unrewarded complexity, such as the silly idea of having a separate thread to update each element of an array
  - Pathologies including starving, racing, and deadlock
- An additional advantage to threads is that they are "light" execution context switches

Summary

- Threading is like stepping into an entirely new world and learning a whole new programming language, or at least a new set of language concepts.
- One of the biggest difficulties with threads occurs because more than one thread might be sharing a resource.
- There's a certain art to the application of threads.
- A significant non-intuitive issue in threading is that, because of thread scheduling, you can typically make your applications run *faster* by inserting calls to **sleep( )** inside **run( )**'s main loop.