Threads of Control

Table of Contents

- Threads of Control
  - What Are Threads?
  - A Simple Thread Example
  - Thread Attributes
    - Thread Body
      - The Clock Applet
    - Thread State
    - Thread Priority
    - Daemon Threads
    - Thread Group
  - Multithreaded Programs
    - Synchronization
    - Deadlock
    - Re−entrant Monitors
  - Summary

Threads of Control

Table of Contents

Threads of Control

Below is three copies of an applet that animates different sorting algorithms. No, this lesson is not about sorting algorithms. But these applets do provide a visual aid to understanding a powerful capability of the Java language––threads.

```
                  Bi-Directional
Bubble Sort        Bubble Sort      Quick Sort
```

Now start each of the applets, one by one, by clicking on them with the mouse. Notice anything? Yes! The applets are running side by side at the same time! Notice anything else? Yes! You can also scroll this page or bring up one of your browser's panels at the same time that the three sorting applets sort their data. All of this is due to the power of *threads*.

What Are Threads?

A thread––sometimes known as an *execution context* or a *lightweight process*––is a single sequential flow of control within a process.

A Simple Thread Example

The following program is a simple Java application that creates and starts two independent threads.

```
class TwoThreadsTest {
    public static void main (String args[]) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

Thread Attributes

To use threads efficiently and without errors you must understand various aspects of threads and the Java runtime system. You need to know how to provide a body for a thread, the life−cycle of a thread, how the runtime system schedules threads, thread groups, and what daemon threads are and how to write them.

Mulithreaded Programs

Up until now all of the sample programs in this lesson have used either one thread or multiple independent threads that run asynchronously. However, it is often useful to use multiple threads that share data and therefore must run synchronously. Typically, programs that use multiple synchronous threads are called multithreaded programs and require special handling.

Summary

This lesson has taken you through the intricacies of Java threads including the life−cycle of a Java thread, scheduling, thread groups, and synchronization. The Java development environment supports multithreaded programs through the language, the libraries, and the runtime system. This summary page highlights all of the features in the Java development environment that supports threads and gives you links to further documentation about those features.

Table of Contents

Threads of Control

What Are Threads?

All programmers are familiar with writing sequential programs. You've probably written a program that displays "Hello World!", or sorts a list of names, or computes a list of prime numbers. These are sequential programs: each has a beginning, an end, a sequence, and at any given time during the runtime of the program there is a single point of execution.

A thread is similar to the sequential programs described above: a single thread also has a beginning, an end, a sequence, and at any given time during the runtime of the thread there is a single point of execution. However, a thread itself is not a program−−it cannot run on its own−−but runs within a program.

Definition: A thread is a single sequential flow of control within a program.

There is nothing new in the concept of a single thread. The real hoopla surrounding threads is not about a single sequential thread, but rather about the use of multiple threads in a single program all running at the same time and performing different tasks.

The HotJava browser is an example of a multithreaded application; within the HotJava browser you can scroll a page while it's downloading an applet or image, play animation and sound concurrently, print a page in the background while you download a new page, or watch three sorting algorithms race to the finish. You are used to life operating in a concurrent fashion...so why not your browser?

Threads of Control

A Simple Thread Example

The simple example shown in full on the previous page defines two classes: SimpleThread and TwoThreadsTest. Let's begin our exploration of the application with the SimpleThread class: a subclass of the Thread class that is provided by the java.lang package.

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

The first method in the SimpleThread class is a constructor that takes a String as its only argument. This constructor is implemented by calling a superclass constructor and is only interesting to us because it sets the Thread's name which is used later in the program.

The next method in the SimpleThread class is the `run()` method. The `run()` method is the heart of any Thread--it's where the action of the Thread takes place. The `run()` method of the SimpleThread class contains a `for` loop that iterates ten times. In each iteration the method display the iteration number and the name of the Thread then sleeps for a random interval between 0 and 1 second. After the loop has finished, the `run()` method prints "DONE!" along with the name of the thread. That's it for the SimpleThread class.

The TwoThreadsTest class provides a `main()` method that creates two SimpleThread threads: one is named "Jamaica" and the other is named "Fiji". (If you can't decide on where to go for vacation you can use this program to help you decide--go to the island whose thread prints "DONE!" first.)

```
class TwoThreadsTest {
    public static void main (String args[]) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

The `main()` method also starts each thread immediately following its construction by calling the

`start()` method. To save you from typing in this program, click here for the source code to the SimpleThread class and here for the source code to the TwoThreadsTest program. Compile and run the program and watch your vacation fate unfold. You should see output similar to the following:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

(Looks like I'm going to Fiji!!) Notice how the output from each thread is intermingled with the output from the other. This is due to the fact that both SimpleThread threads are running concurrently thus both `run()` methods are running at the same time and each thread is displaying its output at the same time as the other.

Try This: Change the main program so that it creates a third thread with the name "Bora Bora". Compile and run the program again. Does this change your island of choice for your vacation? Here's the code for the new main program which we renamed to ThreeThreadsTest.

Keep Going

This page glosses over many of the details of threads such as the `start()` and `sleep()` methods. Don't worry, the next several pages of this lesson explain these concepts and others in detail. The important thing to understand from this page is that a Java program can have many threads, and that those threads can run *concurrently*.

Threads of Control

SimpleThread.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

TwoThreadsTest.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class TwoThreadsTest {
    public static void main (String args[]) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
```

ThreeThreadsTest.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class ThreeThreadsTest {
    public static void main (String args[]) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
        new SimpleThread("Bora Bora").start();
    }
}
```

Thread Attributes

By now, you know are familiar with threads and you've seen a simple Java application that runs two threads concurrently. And you've had an opportunity to watch threads in action using the SystemMonitor applet. This page introduces you several features specific to Java threads and provides you with links to pages that talk about each feature in detail.

Java threads are implemented by the Thread class which is part of the java.lang package. The Thread class implements a system independent definition of Java threads. But under the hood, the actual implementation of concurrent operation is provided by a system−specific implementation. For most programming needs, the underlying implementation doesn't matter; you can ignore the underlying implementation and program to the thread API described in this lesson and the other documentation provided with the Java system.

Thread Body

All of the action takes place in the thread's body−−the thread's `run()` method. You can provide the body to a Thread in one of two ways: by subclassing the Thread class and overriding its `run()` method, or creating a Thread with a Runnable object as its *target*.

Thread State

Throughout its life, a Java thread is in one of several states. A thread's state indicates what the Thread is doing and what it is capable of doing at that time of its life: is it running? is it sleeping? is it dead?

Thread Priority

A thread's priority indicates to the Java thread scheduler when this thread should run in relation to all of the other threads.

Daemon Threads

Daemon threads are those that provide a service for other threads in the system. Any Java thread can be a daemon thread.

Thread Groups

All threads belong to a thread group. ThreadGroup, a java.lang class, defines and implements the capabilities of a group of related threads.

Thread Body

All the action takes place in the thread body which is the thread's `run()` method. After a thread has been created and initialized, the runtime system calls its `run()` method. The code in the `run()` method implements the behaviour for which the thread was created. It's the thread's raison d'etre (reason to be).

Often, a thread's `run()` method is a loop. For example, an animation thread might loop through and display a series of images. Sometimes a thread's `run()` method performs an operation that takes a long time. For example, downloading and playing a sound or a JPEG movie.

There are two different ways that you can provide a customized `run()` method for a Java thread:

1. Subclass the Thread class defined in the java.lang package and override the `run()` method.
   Example: The SimpleThread class used in the the example described in A Simple Thread Example previously in this lesson is an example of this style of using Java threads.
2. Provide a class that implements the Runnable interface, also defined in the java.lang package. Now, when you instantiate a thread (either directly from the Thread class, or from a subclass of Thread), give the new thread a handle to an instance of your Runnable class. This Runnable object provides the `run()` method to the thread.
   Example: The clock applet you see here is an example of using the Runnable interface to provide a `run()` method to a thread. **The Clock Applet** describes the source code for this simple applet.

There are good reasons for choosing either of the two options described above over the other. However, for most cases, the following rule of thumb will guide you to the best option.

Rule of thumb: If your class *must* derive from some other class (the most common example being Applet), then you should use Runnable as described in Option #2.

The Clock Applet

The Clock applet shown here: displays the current time and updates its display every second. You can scroll this page and perform other tasks while the clock continues to update because the code that updates the clock's display runs within its own thread.

This page highlights and explains the source code for the clock applet in detail. In particular, this page describes the code segments that implement the clock's threaded behaviour; it does not describe the code segments that are related to the life cycle of the applet. If you have not written your own applets before or are not familiar with the life cycle of any applet, you may want to take this time to familiarize yourself with the material in The Life Cycle of an Applet before proceeding with this page.

The Decision to Use the Runnable Interface

The Clock applet uses the Runnable interface to provide the `run()` method for its thread. In order to run within a Java–compatible browser, the Clock class had to derive from the Applet class. However, the Clock applet also needs to use a thread so that it can continuously update its display without taking over the process in which it is running. But since the Java language does not support multiple–inheritance, the Clock class could not inherit from Thread as well as from Applet. Thus, the Clock class uses the Runnable interface to provide its threaded behaviour.

Applets are not threads nor do Java–compatible browsers or appletviewers automatically create threads in which to run applets. Therefore, if an applet needs any threads it must create its own. The clock applet needs one thread in which to perform its display updates because it updates its display frequently and the user needs to be able to perform other tasks at the same time the clock is running (such as going to another page, or scrolling this one).

The Runnable Interface

The Clock applet provides a `run()` method for its thread via the Runnable interface. The class definition for the Clock class indicates that the Clock is a subclass of Applet and implements the Runnable interface. If you are not familiar with interfaces review the information at **Missing Page**.

```
class Clock extends Applet implements Runnable {
```

The Runnable interface defines a single method called `run()` that takes no arguments and returns no value. Because the Clock class implements the Runnable interface, it must provide an implementation for the `run()` method as defined in the interface. However, before explaining

the Clock's `run()` method we need to look at some of the other elements of the Clock code first.

Creating the Thread

The application in which an applet is running calls the applet's `start()` method when it loads the applet. The Clock applet creates a Thread named `clockThread` in its `start()` method and starts the thread.

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

First, the `start()` method checks to see if `clockThread` is null. If `clockThread` is null, then the applet is brand new or has been previously stopped and a new thread must be created. Otherwise, the applet is already running. The applet creates a new thread with this invocation:

```
clockThread = new Thread(this, "Clock");
```

Notice that `this`––the Clock applet––is passed in as the first argument to the thread constructor. The first argument to this Thread constructor must implement the Runnable interface and becomes the thread's *target*. When constructed in this way, the thread, `clockThread`, gets its `run()` method from its target Runnable object, in this case, the Clock applet.

The second argument is just a name for the thread.

Stopping the Thread

When you leave the page that displays the Clock applet, the application in which the applet is running calls the applet's `stop()` method. The Clock's `stop()` method stops the thread then sets it to null. This stops the continual updating of the clock.

```
public void stop() {
    clockThread.stop();
    clockThread = null;
}
```

If you revisit the page, the `start()` method is called again, and the clock starts up again with a new thread.

The Run Method

And finally for the Clock's `run()` method. The `run()` method implements the heart of the Clock applet and looks like this:

```
public void run() {
    while (clockThread != null) {
        repaint();
        try {
            clockThread.sleep(1000);
        } catch (InterruptedException e){
        }
    }
}
```

As you saw in the previous section, when the applet is asked to stop, the applet stops the `clockThread` and then sets it to null; this lets the `run()` method know when to stop. Thus the first line of the `run()` method loops until `clockThread` is stopped. Within the loop, the applet repaints itself, and then tells the Thread to sleep for 1 second (1000 milliseconds). An applet's `repaint()` method ultimately calls the applet's `paint()` method which does the actual update of the applet's display area. The Clock applet's `paint()` method gets the current time and draws it to the screen.

```
public void paint(Graphics g) {
    Date now = new Date();
    g.drawString(now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5, 10);
}
```

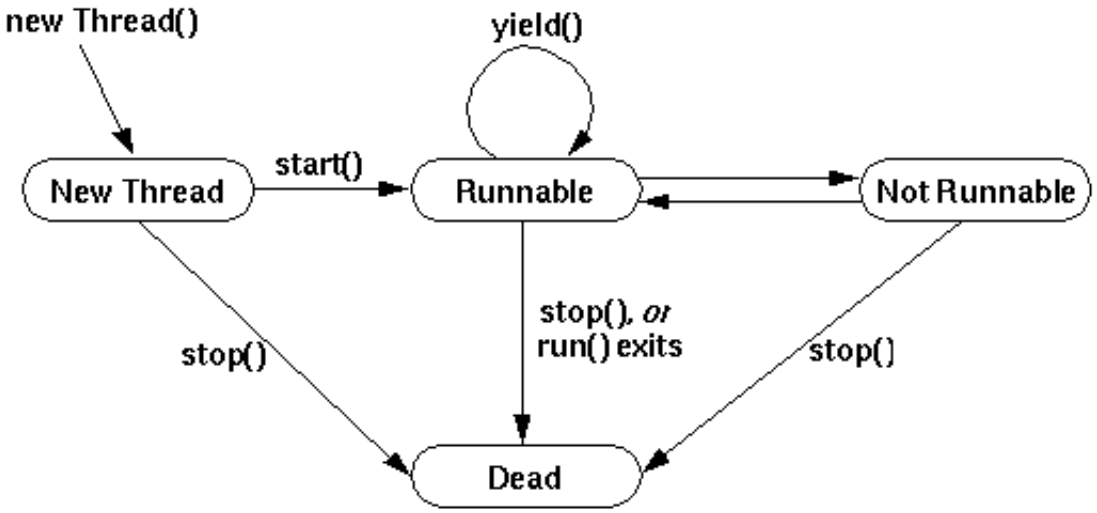
Threads of Control

Clock.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */

import java.awt.Graphics;
import java.util.Date;

public class Clock extends java.applet.Applet implements Runnable {

    Thread clockThread;

    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void run() {
        while (clockThread != null) {
            repaint();
            try {
                clockThread.sleep(1000);
            } catch (InterruptedException e){
            }
        }
    }
    public void paint(Graphics g) {
        Date now = new Date();
        g.drawString(now.getHours() + ":" + now.getMinutes() + ":" + now.getSeconds(), 5, 10);
    }
    public void stop() {
        clockThread.stop();
        clockThread = null;
    }
}
```

Thread State

The following diagram illustrates the various states that a Java thread can be in at any point during its life and which method calls cause a transition to another state. This diagram is not a complete finite state diagram, but rather an overview of the more interesting and common facets of a thread's life. The remainder of this page discusses a Thread's life cycle in terms of its state.



New Thread

The following statement creates a new thread but does not start it thereby leaving the thread in the state labeled "New Thread" in the diagram.

```
Thread myThread = new MyThreadClass();
```

When a thread is in the "New Thread" state, it is merely an empty Thread object. No system resources have been allocated for it yet. Thus when a thread is in this state, you can only start the thread or stop it; calling any other method besides `start()` or `stop()` when a thread is in this state makes no sense and causes an IllegalThreadStateException.

Runnable

Now consider these two lines of code:

```
Thread myThread = new MyThreadClass();
```

```
myThread.start();
```

The `start()` method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's `run()` method. At this point the thread is in the "Runnable" state. This state is called "Runnable" rather than "Running" because the thread may not actually be running when it is in this state. Many computers have a single processor making it impossible to run all "Runnable" threads at the same time. So, the Java runtime system must implement a scheduling scheme that shares the processor between all "Runnable" threads. (See **Thread Priority** for more information about scheduling.) However, for most purposes you can think of the "Runnable" state as simply "Running". When a thread is running−−it's "Runnable" and is the current thread−−the instructions in its `run()` are executing sequentially.

Not Runnable

A thread enters the "Not Runnable" state when one of these four events occur:

- someone calls its `suspend()` method
- someone calls its `sleep()` method
- the thread uses its `wait()` method to wait on a condition variable
- the thread is blocking on I/O.

For example, the bold line in this code snippet

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
    myThread.sleep(10000);
} catch (InterruptedException e){
}
```

puts `myThread` to sleep for 10 seconds (10,000 milliseconds). During those 10 seconds, even if the processor became available `myThread` would not run. After the 10 seconds are up, `myThread` becomes "Runnable" again and now if the processor became available `myThread` would run.

For each of the "entrances" into the "Not Runnable" state listed above, there is a specific and distinct escape route that returns the thread to the "Runnable" state. An escape route only works for its corresponding "entrance". For example, if a thread has been put to sleep, then the specified number of milliseconds must elapse before the thread becomes "Runnable" again. Calling `resume()` on a sleeping thread has no effect.

The following indicates the escape route for every entrance into the "Not Runnable" state.

- If a thread has been put to sleep, then the specified number of milliseconds must elapse.
- If a thread has been suspended, then someone must call its `resume()` method.
- If a thread is waiting on a condition variable, whatever object owns the variable must relinquish it by calling either `notify()` or `notifyAll()`.
- If a thread is blocked on I/O, then the specified I/O command must complete.

Dead

A thread can die in two ways: either from natural causes, or by being killed (stopped). A thread dies naturally when its `run()` method exits normally. For example, the `while` loop in this method is a finite loop−−it will iterate 100 times and then exit.

```
public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}
```

A thread with this `run()` method will die naturally after the loop and the `run()` method completes.

You can also kill a thread at any time by calling its `stop()` method. This code snippet

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
    Thread.currentThread().sleep(10000);
} catch (InterruptedException e){
}
myThread.stop();
```

creates and starts `myThread` then puts the current thread to sleep for 10 seconds. When the current thread wakes up, the bold line in the code segment kills `myThread`.

The `stop()` method throws a ThreadDeath object at the thread to kill it. Thus when a thread is killed in this manner it dies asynchronously. The thread will die when it actually receives the ThreadDeath exception.

Applets often use the `stop()` method to kill all of its threads when the Java−compatible browser in which it is running tells the applet to stop (when the user changes pages for example). See Threads in Applets for more information.

IllegalThreadStateException

The runtime system throws an IllegalThreadStateException when you call a method on a thread and that thread's state does not allow for that method call. For example, IllegalThreadStateException is thrown when you call `suspend()` on a thread that is not "Runnable".

As shown in the various examples of threads so far in this lesson, when you call a thread method that can throw an exception, you must either catch and handle the exception, or declare that the calling method throws the uncaught exception.

The isAlive() Method

And a final word about thread state: the programming interface for the Thread class includes a method called `isAlive()`. The `isAlive()` returns true if the thread has been started and not stopped. Thus, if the `isAlive()` method returns **false** you know that the thread is either a "New Thread" or "Dead". If the `isAlive()` method returns **true**, you know that the thread is either "Runnable" or "Not Runnable". You cannot differentiate between a "New Thread" and a "Dead" thread; nor can you differentiate between a "Runnable" thread and a "Not Runnable" thread.

See Also

api java.lang.Thread

api java.lang.IllegalThreadStateException

api java.lang.ThreadDeath

Threads of Control

Thread Priority

Previously in this lesson, we've claimed that threads run concurrently. While conceptually this is true, in practice it isn't. Most computer configurations have a single CPU, so threads actually run one at a time in such a way as to simulate concurrency. The execution of multiple threads on a single CPU, in some order, is called *scheduling*. The Java runtime supports a very simple, deterministic scheduling algorithm known as *fixed priority scheduling*. This algorithm schedules threads based on their *priority* relative to other "Runnable" threads.

When a Java thread is created, it inherits its priority from the thread that created it. You can also modify a thread's priority at any time after its creation using the `setPriority()` method. Thread priorities range between MIN_PRIORITY and MAX_PRIORITY (constants defined in class Thread). At any given time, when multiple threads are ready to be executed, the runtime system chooses the "Runnable" thread with the highest priority for execution. Only when that thread stops, yields, or becomes "Not Runnable" for some reason, will a lower priority thread start executing. If there are two threads of the same priority waiting for the CPU, the scheduler chooses them in a round−robin fashion.

The Java runtime system's thread scheduling algorithm is also *preemptive*. If at any time a thread with a higher priority than all other "Runnable" threads becomes "Runnable", the runtime system chooses the new higher priority thread for execution. The new higher priority thread is said to preempt the other threads.

The Java runtime system's thread scheduling scheme can be summed up with this simple rule:

Rule: At any given time, the highest priority runnable thread is running.

The 400,000 Micron Thread Race

This Java source code implements an applet that animates a race between two "runner" threads with different priorities. When you click the mouse down over the applet, it starts the two runners. The top runner, labelled "1", has a priority of 1 (the lowest possible thread priority in the Java system). The second runner, labelled "2", has a priority of 2.

Try This: Click over the applet below to start the race.

This is the `run()` method for both runners.

```
public int tick = 1;
public void run() {
```

```
        while (tick < 400000) {
            tick++;
        }
    }
```

This `run()` method simply counts from 1 to 400,000. The instance variable `tick` is public because the applet uses this value to determine how far the runner has progressed (how long its line is).

In addition to the two runner threads, this applet also has a third thread that handles the drawing. The drawing thread's `run()` method contains an infinite loop; during each iteration of the loop it draws a line for each runner (whose length is computed from the runner's `tick` variable), and then sleeps for 10 milliseconds. The drawing thread has a thread priority of 3−−higher than either runner. So, whenever the drawing thread wakes up after 10 milliseconds, it becomes the highest priority thread, preempting whichever runner is currently running and draws the lines. Thus you can see the lines inch their way across the page

As you can see, this is not a fair race because one runner has a higher priority than the other. Each time the drawing thread yields the CPU by going to sleep for 10 milliseconds, the scheduler chooses the highest priority runnable thread to run; in this case, it's always the runner labelled "2". Here is another version of the applet that implements a "fair race", that is, both of the runners have the same priority and they have an equal chance of being chosen to run.

Try this: Again, click down with the mouse to start the race.

In this race, each time the drawing thread yields the CPU by going to sleep, there are two Runnable threads of equal priority−−the runners−−waiting for the CPU; the scheduler must choose one of the threads to run. In this situation, the scheduler chooses the next thread to run in a round−robin fashion.

Selfish Threads

The Runner class used in the races above actually implements "socially−impaired" thread behaviour. Recall the `run()` method from the Runner class used in the races above:

```
public int tick = 1;
public void run() {
    while (tick < 400000) {
        tick++;
    }
}
```

The `while` loop in the `run()` method is in a tight loop. That is to say, once the scheduler chooses a thread with this thread body for execution, the thread never voluntarily relinquishes control of the CPU−−the thread continues to run until the `while` loop terminates naturally or until the thread is preempted by a higher priority thread.

In some situations, having "selfish" threads doesn't cause any problems because a higher priority

thread preempts the selfish one (just as the drawing thread in the RaceApplet preempts the selfish runners). However, in other situations, threads with CPU−greedy `run()` methods, such as the Runner class, can take over the CPU and cause other threads to have to wait for a long time before getting a chance to run.

Time−Slicing

Some systems fight selfish thread behaviour with a strategy known as *time−slicing*. Time−slicing comes into play when there are multiple "Runnable" threads of equal priority and those threads are the highest priority threads competing for the CPU. For example, this stand−alone Java program (which is based on the RaceApplet above) creates two equal priority selfish threads that have the following `run()` method.

```
public void run() {
    while (tick < 400000) {
        tick++;
        if ((tick % 50000) == 0) {
            System.out.println("Thread #" + num + ", tick = " + tick);
        }
    }
}
```

This `run()` contains a tight loop that increments the integer `tick` and every 50,000 ticks prints out the thread's identifier and its `tick` count.

When running this program on a time−sliced system, you will see messages from both threads intermingled with one another. Like this:

```
Thread #1, tick = 50000
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #1, tick = 250000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 400000
```

This is because a time−sliced system divides the CPU into time slots and iteratively gives each of the equal−and−highest priority threads a time slot in which to run. The time−sliced system will continue to iterate through the equal−and−highest priority threads allowing each one a bit of time to run until or more of them finish or until a higher priority preempts them. Notice that time−slicing makes no guarantees as to how often or in what order threads are scheduled to run.

When running this program on a non−time−sliced system, however, you will see messages from one thread finish printing before the other thread ever gets a chance to print one message. Like this:

```
Thread #0, tick = 50000
Thread #0, tick = 100000
Thread #0, tick = 150000
Thread #0, tick = 200000
Thread #0, tick = 250000
Thread #0, tick = 300000
Thread #0, tick = 350000
Thread #0, tick = 400000
Thread #1, tick = 50000
Thread #1, tick = 100000
Thread #1, tick = 150000
Thread #1, tick = 200000
Thread #1, tick = 250000
Thread #1, tick = 300000
Thread #1, tick = 350000
Thread #1, tick = 400000
```

This is because a non−time−sliced system chooses one of the equal−and−highest priority threads to run and allows that thread to run until it relinquishes the CPU (by sleeping, yielding, finishing its job) or until a higher priority preempts it.

Try this: Compile and run the RaceTest and SelfishRunner classes on your computer. Can you tell if you have a time−sliced system?

As you can imagine, writing CPU−intensive code can have negative repercussions on other threads running in the same process. In general, you should try to write "well−behaved" threads that voluntarily relinquish the CPU periodically and give other threads an opportunity to run. In particular, you should never write Java code that relies on time−sharing−−this will practically guarantee that your program will give different results on a different computer system.

A thread can voluntarily yield the CPU (without going to sleep or some other drastic means) by calling the `yield()` method. The `yield()` method gives other threads of the same priority a chance to run. If there are no equal priority threads in the "Runnable" state, then the yield is ignored.

Try this: Rewrite the SelfishRunner class to be a PoliteRunner by calling the `yield()` method from the `run()` method. Be sure to modify the main program to create PoliteRunners instead of SelfishRunners. Compile and run the new classes on your computer. Now isn't that better?

Summary

- In most configurations, there is only one CPU, thus threads must share the CPU with other threads. The execution of multiple threads on a single CPU, in some order, is called scheduling. The Java runtime supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling.

- Each Java thread is given a numeric priority, between MIN_PRIORITY and MAX_PRIORITY (constants defined in class Thread). At any given time, when multiple threads are ready to be executed, the thread with the highest priority will be chosen for execution. Only when that thread stops, or is suspended for some reason, will a lower priority thread start executing.
- Scheduling of the CPU is fully preemptive. If a thread with a higher priority than the currently executing thread needs to execute, the higher priority thread is immediately scheduled.
- The Java runtime will not preempt the currently running thread for another thread of the same priority. In other words, the Java runtime does not time−slice. However, the system implementation of threads underlying the Java Thread class may support time−slicing. Do not write code that relies on time−slicing.

  If the currently running thread yields the CPU (i.e. allows another thread to execute by calling the `yield()`), then the scheduler implements a simple non−preemptive round−robin scheduling order.
- In addition, a given thread may, at any time, give up its right to execute by calling the `yield()` method. Threads can only yield the CPU to other threads of the same priority−−attempts to yield to a lower priority thread are ignored.

Threads of Control

RaceApplet.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
import java.awt.*;

public class RaceApplet extends java.applet.Applet implements Runnable {

    final static int NUMRUNNERS = 2;
    final static int SPACING = 20;

    Runner runners[] = new Runner[NUMRUNNERS];

    Thread updateThread;

    public void init() {
        String raceType = getParameter("type");
        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i] = new Runner();
            if (raceType.compareTo("unfair") == 0)
                runners[i].setPriority(i+1);
            else
                runners[i].setPriority(2);
        }
        if (updateThread == null) {
            updateThread = new Thread(this, "Thread Race");
            updateThread.setPriority(NUMRUNNERS+1);
        }
    }

    public boolean mouseDown(java.awt.Event evt, int x, int y) {
        if (!updateThread.isAlive())
            updateThread.start();
        for (int i = 0; i < NUMRUNNERS; i++) {
            if (!runners[i].isAlive())
                runners[i].start();
        }
        return true;
    }
```

```
    public void paint(Graphics g) {
        g.setColor(Color.lightGray);
        g.fillRect(0, 0, size().width, size().height);
        g.setColor(Color.black);
        for (int i = 0; i < NUMRUNNERS; i++) {
            int pri = runners[i].getPriority();
            g.drawString(new Integer(pri).toString(), 0, (i+1)*SPACING);
        }
        update(g);
    }

    public void update(Graphics g) {
        for (int i = 0; i < NUMRUNNERS; i++) {
            g.drawLine(SPACING, (i+1)*SPACING, SPACING + (runners[i].tick)/1000, (i+1)*SPACING);
        }
    }

    public void run() {
        while (updateThread != null) {
            repaint();
            try {
                updateThread.sleep(10);
            } catch (InterruptedException e) {
            }
        }
    }

    public void stop() {
        for (int i = 0; i < NUMRUNNERS; i++) {
            if (runners[i].isAlive()) {
                runners[i].stop();
                runners[i] = null;
            }
        }
        if (updateThread.isAlive()) {
            updateThread.stop();
            updateThread = null;
        }
    }
}
```

Runner.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class Runner extends Thread {

    public int tick = 1;

    public void run() {
        while (tick < 400000) {
            tick++;
        }
    }
}
```

RaceTest.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class RaceTest {

    final static int NUMRUNNERS = 2;

    public static void main(String args[]) {

        SelfishRunner runners[] = new SelfishRunner[NUMRUNNERS];

        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i] = new SelfishRunner(i);
            runners[i].setPriority(2);
        }
        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i].start();
        }
    }
}
```

SelfishRunner.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class SelfishRunner extends Thread {

    public int tick = 1;
    public int num;

    SelfishRunner(int num) {
        this.num = num;
    }

    public void run() {
        while (tick < 400000) {
            tick++;
            if ((tick % 50000) == 0) {
                System.out.println("Thread #" + num + ", tick = " + tick);
            }
        }
    }
}
```

PoliteRunner.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class PoliteRunner extends Thread {

    public int tick = 1;
    public int num;

    PoliteRunner(int num) {
        this.num = num;
    }

    public void run() {
        while (tick < 400000) {
            tick++;
            if ((tick % 50000) == 0) {
                System.out.println("Thread #" + num + ", tick = " + tick);
                yield();
            }
        }
    }
}
```

RaceTest2.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class RaceTest2 {

    final static int NUMRUNNERS = 2;

    public static void main(String args[]) {

        PoliteRunner runners[] = new PoliteRunner[NUMRUNNERS];

        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i] = new PoliteRunner(i);
            runners[i].setPriority(2);
        }
        for (int i = 0; i < NUMRUNNERS; i++) {
            runners[i].start();
        }
    }
}
```

Daemon Threads

Any Java thread can be a *daemon* thread. Daemon threads are service providers for other threads or objects running in the same process as the daemon thread. For example, the HotJava browser has a daemon thread, named *Background Image Reader*, that reads images from the filesystem or the network for any object or thread that needs an image.

Daemon threads are typically independent threads within an application that provide services for other objects within that same application. The `run()` method for a daemon thread is typically an infinite loop that waits for a service request.

When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when there are only daemon threads remaining, there is no other thread for which a daemon thread can provide a service.

To specify that a thread is a daemon thread call the `setDaemon()` method with a boolean parameter that is true. To determine if a thread is a daemon thread use the accessor method `isDaemon()`.

[PENDING: write an example]

Thread Group

[PENDING: this page is under construction]

See also

api java.lang.ThreadGroup

Multithreaded Programs

Synchronization

Often, threads need to share data. For example, suppose you have a thread that writes data to a file while, at the same time, another thread is reading data from that same file. When your threads need to share information you need to synchronize the threads to get the desired results.

Deadlock

Deadlock occurs when two threads are waiting for some condition to change that is precluded from changing because of all threads that can change the condition are waiting. This page uses the dining philosophers to illustrate the problem of deadlock, and discusses various ways to avoid it.

Re−entrant Monitors

Java monitors are re−entrant.

Volatile

[PENDING: This technology is under construction.]

Synchronization

So far, this lesson has contained examples with independent, asynchronous threads. That is, each thread contained all of the data and methods required for its execution and didn't require any outside resources or methods. In addition, the threads in those examples ran at their own pace without concern over the state or activities of any other concurrently running threads.

However, there are many interesting situations where separate concurrently running threads do share data and must consider the state and activities of those other threads. One such set of programming situations are known as Producer/Consumer scenarios where the Producer generates a stream of data which then is consumed by a Consumer.

For example, you can imagine a Java application where one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. Or, as you type characters on the keyboard, the producer thread places key events in an event queue and the consumer thread reads the events from the same queue. Both of these examples use concurrent threads that share a common resource: a file, an event queue. And because the threads share a common resource, they must be synchronized in some way.

This lesson teaches you about Java thread synchronization through a simple Producer/Consumer example.

Producer/Consumer Example

The Producer generates integers ranging from 0 to 9, stores it in a "CubbyHole" object, prints the generated number, and (just to make the synchronization problem more interesting) the Producer sleeps for a random amount of time between 0 and 100 milliseconds.

```
class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
```

```
            }
        }
    }
}
```

The Consumer, being ravenous, consumes all integers from the CubbyHole (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

```
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number + " got: " + value);
        }
    }
}
```

The Producer and Consumer in this example share data through a common `CubbyHole` object. And you will note that neither the Producer nor the Consumer make any effort whatsoever to ensure that the Consumer is getting each value produced once and only once. The synchronization between these two threads actually occurs at a lower level, within the `get()` and `put()` methods of the CubbyHole object. However, let's assume for a moment that no synchronization between these two threads and talk about the potential problems that might arise in that situation.

One problem that might arise if the Producer and Consumer threads make no arrangements to run synchronously occurs when the Producer is quicker than the Consumer and generates two numbers before the Consumer has a chance to consume the first one. Thus the Consumer would skip a number. Part of the output might look like this:

```
    . . .

Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5

    . . .
```

Another problem that might arise is when the Consumer is quicker than the Producer and consumes the same value twice. In this situation, the Consumer would print the same value twice

and might produce output that looked like this:

```
     . . .

Producer #1 put: 4
Consumer #1 got: 4
Consumer #1 got: 4
Producer #1 put: 5

     . . .
```

Either way, the result is wrong. You want the Consumer to get each integer produced by the Producer exactly once. Problems, such as those just described, that arise from multiple, asynchronously executing threads trying to a single object at the same time and getting the wrong result, are called *race conditions*.

To prevent race conditions in our Producer/Consumer example, the storage of a new integer into the CubbyHole by the Producer must be synchronized with the retrieval of an integer from the CubbyHole by the Consumer. The Consumer must consume each integer exactly once. Objects such as the CubbyHole, which are shared between two threads and whose accesses must be synchronized, are called *condition variables*. The Java language allows you to synchronize threads around a condition variable through the use of *monitors*.

Monitors

The Java language and runtime system support thread synchronization through the use of *monitors* which were first outlined in C. A. R. Hoare's article *Communicating Sequential Processes* (*Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666–677). In general, a monitor is associated with a specific data item (the condition variable) and functions as a lock on that data. When a thread holds the monitor for some data item, other threads are locked out and cannot inspect or modify the data.

The code segments within a program that make it possible for separate, concurrent threads to access the same data items are known as *critical sections*. In the Java language, you identify critical sections in your program with the `synchronized` keyword.
Note: Generally, critical sections in Java programs are methods. You can mark smaller code segments as synchronized. However, this violates object–oriented paradigms and leads to confusing code that is difficult to debug and maintain. For the majority of your Java programming purposes, it's best to use `synchronized` only at the method level.

In the Java language, a unique monitor is associated with every object that has a synchronized method. The CubbyHole class for the Producer/Consumer example introduced above has two synchronized methods: the `put()` method used to change the value in the CubbyHole and the `get()` method used to retrieve the current value.

```
class CubbyHole {
    private int seq;
```

```
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        available = false;
        return seq;
    }

    public synchronized void put(int value) {
        seq = value;
        available = true;
        notify();
    }
}
```

In addition, the `CubbyHole` has two private variables: `seq` which is the current contents of the CubbyHole, and the boolean variable `available` which indicates whether the CubbyHole contents can be retrieved. When `available` is true the Producer has just put a new value in the CubbyHole and the Consumer has not yet consumed it. The Consumer can only consume the value in the CubbyHole when `available` is true.

Because CubbyHole has synchronized methods, the Java language provides a unique monitor for each instance of CubbyHole, including the one shared by the Producer and the Consumer. Whenever the Producer calls the CubbyHole's `put()` method, the Producer acquires the monitor for the CubbyHole thereby preventing the Consumer from calling the CubbyHole's `get()` method.

```
public synchronized void put(int value) {
        // monitor has been acquired by the Producer
    seq = value;
    available = true;
    notify();
        // monitor is released by the Producer
}
```

When the `put()` method returns, the Producer releases the monitor thereby unlocking the CubbyHole.

Conversely, whenever the Consumer calls the CubbyHole's `get()` method, the Consumer acquires the monitor for the CubbyHole thereby preventing the Producer from calling the `put()` method.

```
public synchronized int get() {
        // monitor has been acquired by the Consumer
    while (available == false) {
        try {
            wait();
```

```
            } catch (InterruptedException e) {
            }
        }
        available = false;
        return seq;
            // monitor is released by the Consumer
    }
```

The acquisition and release of a monitor is done atomically by the Java runtime system. This ensures that race conditions cannot occur in the underlying implementation of the threads and ensures data integrity.

You might have noticed a potential problem in CubbyHole's `put()` and `get()` methods. At the beginning of the `get()` method, if the value in the CubbyHole is not available (that is, the Producer has not generated a new number since the last time the Consumer consumed it) then the Consumer waits for the Producer to put a new value into the CubbyHole. So, the question arises−−how can the Producer put a new value into the CubbyHole, if the Consumer holds the monitor (the Consumer holds the CubbyHole's monitor because it's within the synchronized method `get()`)?

Well, the designers of the Java language thought of this too. When the thread enters the `wait()` method, the monitor is released atomically, and when the thread exits the `wait()` method, the monitor is acquired again. This gives the Producer the opportunity to acquire the monitor and put a new value into the CubbyHole.

The Main Program

Here's a small stand−alone Java application that creates a CubbyHole object, one Producer, one Consumer, and then starts both the Producer and the Consumer.

```
class ProducerConsumerTest {
    public static void main(String args[]) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

The Output

Here's the output of ProducerConsumerTest.

```
Producer #1 put: 0
Consumer #1 got: 0
Producer #1 put: 1
Consumer #1 got: 1
```

```
Producer #1 put: 2
Consumer #1 got: 2
Producer #1 put: 3
Consumer #1 got: 3
Producer #1 put: 4
Consumer #1 got: 4
Producer #1 put: 5
Consumer #1 got: 5
Producer #1 put: 6
Consumer #1 got: 6
Producer #1 put: 7
Consumer #1 got: 7
Producer #1 put: 8
Consumer #1 got: 8
Producer #1 put: 9
Consumer #1 got: 9
```

Try this: Remove the lines that are shown in bold in the listing of the CubbyHole class shown above. Recompile the program and run it again. What happened? Because no explicit effort has been made to synchronize the Producer and Consumer threads, the Consumer consumes with reckless abandon and gets a whole bunch of zeros instead of getting each integer between 0 and 9 exactly once.

Threads of Control

Producer.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Consumer.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number + " got: " + value);
        }
    }
}
```

CubbyHole.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class CubbyHole {
    private int seq;         // this is the condition variable.
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        available = false;
        return seq;
    }

    public synchronized void put(int value) {
        seq = value;
        available = true;
        notify();
    }
}
```

ProducerConsumerTest.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
class ProducerConsumerTest {
    public static void main(String args[]) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

Deadlock

[PENDING: this page is under construction]

Threads of Control

Re–entrant Monitors

[PENDING: this page is under construction]

Threads of Control

Summary

The pages of this lesson have provided a lot of information about using threads in the Java development environment. Threads are supported by various components of the Java development environment and it can be confusing where to look for the features that you need. This page summarizes where in the Java environment you can find various classes, methods, and language features that participate in the Java threads story.

Package Support of Threads

java.lang.Thread api

In the Java development enviroment, threads are objects that derive from java.lang's Thread class. The Thread class defines and implements Java threads. You can subclass the Thread class to provide your own thread implementations or you can use the Runnable interface.

java.lang.Runnable api

The Java language library also defines the Runnable interface which allows any arbitrary object to provide the body (the `run()` method) for a Thread.

java.lang.Object api

The base level class, Object, defines three methods you can use to synchronize methods around a condition variable: `wait()`, `notify()`, and `notifyAll()`.

java.lang.ThreadGroup api

All threads belong to a thread group which typically contains related threads. The ThreadGroup class in the java.lang package implements groups of threads.

java.lang.Death api

A thread is normally killed by throwing a ThreadDeath object at it. It is rare that any thread need catch ThreadDeath to do any cleaning up before actually dying.

Language Support of Threads

The Java language has two keywords related to the synchronization of threads: `volatile` (which is not yet implemented) and `synchronized`. Both of these language features help ensure the integrity of data that is shared between two concurrently running threads. Multithreaded Programs discusses thread synchronization issues.

Runtime Support of Threads

The Java runtime system contains the scheduler that is responsible for running all the existing threads. The Java scheduler uses a fixed priority scheduling algorithm which boils down to this simple rule:

Rule: At any give time, the highest priority runnable thread is running.

Threads of Control