

# Chapter 8: Threads

# **Table of Contents**

Objectives.....	3
Section 1: Introduction .....	4
Threads .....	5
Thread Requirements .....	6
Creating Threads .....	7
Threads Using Inheritance .....	8
Runnable Interface .....	9
Controlling Threads .....	10
sleep Demonstration.....	12
Thread Scheduling .....	13
Thread Synchronization .....	14
wait and notify Methods .....	15
Multiple Thread Example .....	16
sleep, suspend, and resume Methods.....	17
Thread Priorities.....	18
Summary .....	19

## Objectives

*After completing this unit you will be able to:*

- Understand how threads are supported in Java
- Use inheritance or an interface to implement threads
- Control the thread's creation and termination
- Use the **sleep** method
- Define and use synchronization
- Use the wait and notify methods to control the execution of a thread

# Section 1: Introduction

## Threads

Threads are concurrent execution paths within an application. Threads can be used to make programs more responsive, therefore more user friendly. Java threads are guaranteed to run even on non-multitasking operating systems. They are used to support a number of different types of applications including:

- Word processor background printing
- Lengthy calculations
- Animation
- Any process that either takes a long time or can be done in its own time
- Processes that can be done in parallel

## Thread Requirements

A thread requires:

- Code to execute
- Data to execute against
- CPU to support the execution

In Java, a class provides the code and data. The code is in the methods and the data is the instance variables. A Thread class provides a virtual CPU. Each thread will have its own virtual or actual CPU and corresponding program stack.

The underlying operating system effects how threads are implemented in Java. The operating system often supplies the much of the functionality of the virtual machines. As a result the behavior of threads will vary some what depending on the operating system that Java is using. These differences will be discussed as they become relevant.

## Creating Threads

There are two techniques that a class uses to create a thread:

- Extend the Thread class
- Implement the Runnable interface

Extending the Thread class is not normally used. Since Java does not support multiple inheritance between classes, if inheritance is used, the class cannot be derived from another class. When inheriting from the Thread class, the **run** method is overridden. This method has no parameters and returns void.

The preferred technique is by implementing the Runnable interface. The Runnable interface consists of a single method **run** which has the same signature as that of the Thread class's **run** method.

Regardless of whether the thread is created through inheritance or by implementing an interface, the thread's code is found in a **run** method or in the methods called by the **run** method. A thread will terminate when the **run** method exits.

## Threads Using Inheritance

The thread can extend the Thread class and override the **run** method. The Thread class's **run** method does nothing. Normally, the derived class will provide thread functionality by overriding this method. To start the thread, an instance of the derived class is created and the **start** method is executed against this object.

Extending the Thread class is illustrated below. The application repeatedly echoes the string used in the class's constructor to standard output. Note that the **run** method consists of a single loop that continues indefinitely. This is not an uncommon practice but does require that the thread be stopped.

```
class EchoDemo1 extends Thread {
    String echoString;

    public static void main(String args[]) {
        EchoDemo1 e = new EchoDemo1("Thread is executing");
        e.start();
    }

    public EchoDemo1(String initialString) {
        echoString = initialString;
    }

    public void run() {
        while(true)
        {
            System.out.println(echoString);
        }
    }
}
```

## Runnable Interface

The Runnable interface consists of a single method **run**:

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

To create a thread using this interface, the new thread implements the Runnable interface. Next, two object are created.

- The first object is an instance of the new thread class
- The second object is an instance of the Thread class

The constructor of the Thread class will take an object that implements the Runnable interface. The new thread object is used as the argument to the constructor. The **start** method is then applied to the Thread object and the thread starts.

```
class EchoDemo2 implements Runnable {  
    String echoString;  
  
    public static void main(String args[])  
    {  
        EchoDemo2 e = new EchoDemo2("Thread is executing");  
        Thread t = new Thread(e);  
        t.start();  
    }  
  
    public EchoDemo2(String initialString)  
    {  
        echoString = initialString;  
    }  
  
    public void run() {  
        while(true) {  
            System.out.println(echoString);  
        }  
    }  
}
```

## Controlling Threads

To start a thread, the **start** method is used. A thread is terminated with the **stop** method. However, this method was deprecated in JDK 1.2. No new method was introduced to replace the **stop** method. Instead, the recommended approach is to modify the body of the **run** method so that it incorporates a boolean flag. While the flag is true, the thread runs. When the flag is set to false, the thread stops.

```
boolean threadIsRunning;
...

public void run() {
    while(threadIsRunning) {
        count++;
    }
}
```

Methods can be added to the class to stop and start the thread.

```
public StartDemo () {
    threadIsRunning = true;
}

public void stop() {
    threadIsRunning = false;
}
```

The StartStopDemo class demonstrates this technique.

```
import java.io.*;

class StartStopDemo implements Runnable {
    String echoString;
    boolean threadIsRunning;

    static int count = 0;
    static boolean counting = true;
```

```
public static void main(String args[]) throws IOException {
    String command;

    StartStopDemo e = new StartStopDemo();
    Thread t = new Thread(e);
    t.start();

    while (counting) {
        System.out.print("Command: ");
        command = readString();
        if (command.equals("stop")) {
            e.stop();
            System.out.println("The final value of count is: " + count);
            counting = false;
        } else {
            System.out.println(
                "Invalid command:" + command + ":");
            System.out.println("The current value of count is: " +
                count);
        }
    }
}

public StartStopDemo () {
    threadIsRunning = true;
}

public void stop() {
    threadIsRunning = false;
}

public void run() {
    while(threadIsRunning) {
        count++;
    }
}
}
```

## sleep Demonstration

A thread can suspend itself for a fixed period of time using the **sleep** method. This method suspends the thread's execution for a specified period of time.

```
public static void sleep(long millis) throws InterruptedException
```

The static **currentThread** method returns a reference to the current thread. The **sleep** method in the next example suspends the thread for 3 seconds.

```
class SleepDemo implements Runnable {
    String echoString;
    static int count = 0;
    static boolean counting = true;

    public static void main(String args[]) {
        SleepDemo e = new SleepDemo();
        Thread t = new Thread(e);
        t.start();
    }

    public void run() {
        System.out.println("Starting thread");
        try {
            System.out.println("Waiting !");
            Thread.currentThread().sleep(3000);
            System.out.println("Done!");
        }
        catch (InterruptedException e) {};
        System.out.println("Thread terminating");
    }
}
```

## **Thread Scheduling**

The scheduler determines the real-time ordering of threads within a system. The scheduler will use either non-preemptive or preemptive scheduling. Non-preemptive scheduling allows the current thread to run to completion and does not interrupt the thread. The thread must explicitly yield control for another task to gain control of the computer.

Preemptive scheduling will preempt or interrupt a thread. Typically, threads are allocated a time slice. This gives all threads a chance to execute and complete in a more predictable fashion. Applications are easier to program and most popular operating systems use this approach.

Current Java specification does not specify how the scheduler should be implemented. Future releases of Java may specify how the scheduler should be implemented.

## Thread Synchronization

In a multithreaded application, it is possible for more than one thread to share the same data or resource. When this happens, the shared data can be left in a corrupted state. For example, two threads may need to write information to the same printer. If they are allowed to both write at the same time, the output will become interspersed. It is important that one thread be granted exclusive access to the printer resource until it is through printing. The same is true of same data.

The code that access shared data or a share resource is called the critical section. This is the code that will use the resource or modify the data. In order for a thread to gain access to the resource, the resource must be synchronized. The details of how one thread gains access to the resource and others are denied access is largely hidden from the programmer. All that he has to do is to declare the resource as synchronized. This is done using the *synchronized* keyword.

The synchronized keyword is applied against a statement or a method:

```
synchronized (someObject) {  
    statements  
}  
  
public synchronized someMethod() { ... }
```

A thread that needs to execute a critical section will first attempt to gain a lock on the object. The synchronization keyword hides the actual process of gaining a lock. If another thread already has a lock then the requesting thread is block until the object is released.

More than one critical section may be acting on the same object. For example, there may be one sequence of code that reads from a file and another sequence of code that writes to a file. If one thread is reading and anther thread is writing to the *same* file and the read and write code sequences are synchronized, then one of the threads will be blocked.

When using the synchronized keyword with a block of code, it is clear which object is being synchronized. In the example below it is someObject:

```
synchronized (someObject) {  
    statements  
}
```

When a method is synchronized, the object is the object that the method is running against.

## wait and notify Methods

A producer/consumer relationship is a fairly common paradigm in computer science. A producer will produce some object and then make it available for a consumer. The producer may add an item to a queue and the consumer will get the object from the queue. If the consumer is removing objects from the queue faster than the producer puts them in then we may want to suspend the consumer until something is ready.

This can be done using the **wait** and **notifyAll** methods. When the consumer finds that the queue is empty it can execute the **wait** method. It will then be suspended until the **notify** or **notifyAll** method is executed against the shared object. If the producer adds an object to the queue and then executes the **notifyAll** method, the consumer will awake and try to get access to the queue. The producer and consumer can easily communicate in this manner.

It is possible that there may be more than one producer and more than one consumer all sharing the same queue. When the **notify** method is executed, then one of the suspended threads will wake up and proceed. The one that will wake up is not determinate and could be any of the suspended threads.

When the **notifyAll** method is used in this example, all of the waiting threads are awakened and then compete to acquire the queue. No one thread has an advantage over another.

## Multiple Thread Example

The use of multiple threads is illustrated below. The **currentThread** method returns a reference to the currently running thread. When a thread is created, a name can be assigned to the thread through its constructor. The name can later be retrieved using the **getName** method.

```
class SimpleThread implements Runnable {
    public void run () {
        while (true) {
            System.out.println (Thread.currentThread().getName());
        } // while
    } // of run
} // of class Simple Thread
```

```
public class SchedulerTest {
    public static void main(String args[]) {
        SimpleThread aThread = new SimpleThread ();

        new Thread (aThread, "First thread").start();
        new Thread (aThread, "Second thread").start();
    } // of main
} // of class Scheduler Test
```

## sleep, suspend, and resume Methods

Sometimes it is desirable to be able to suspend the execution of a thread for a while and then resume execution of the thread. The **sleep** method does this but it is for a fixed length of time. To suspend and then resume a thread for something other than a fixed length of time the **suspend** and **resume** methods are available. However, they have been deprecated in Java 1.2.

The recommended way of suspending and resuming threads is to use the same technique used to replace the **stop** method. That is, create a boolean flag that determines if the thread is suspended or not. Use the variable to control the execution of a while loop in the run method. Methods are then added to set or reset this flag.

In the next example, the flag, *suspended*, is used to indicate whether the thread has been suspended or not. When set to true, the thread is suspended. The flag has been declared as volatile which means that the compiler should not attempt to optimize code that modifies the flag in such a manner that would affect its value. **wait** is used to actually suspend the thread. In a *resume* method, a notify method would be used to wake the thread up.

```
private boolean volatile suspended;

public void run() {
    while (threadIsRunning) {
        try {
            if (suspended) {
                synchronized (this) {
                    while (suspended)
                        wait();
                }
            }
            // Thread body
        }
        catch (InterruptedException e){
        }
    }
}
```

## Thread Priorities

In a preemptive operating system, threads are assigned priorities and the highest priority is given in preference over lower priority threads. The **setPriority** method and **getPriority** methods are used to assign and retrieve the priority of a thread respectively. The argument of the setPriority method accepts an integer between 1 and 10 inclusively. 10 is the highest priority and 1 is the lowest priority.

Java has declared three constants for use with priorities:

```
public static final int  MAX_PRIORITY = 10;
public static final int  MIN_PRIORITY = 1;
public static final int  NORM_PRIORITY = 5;
```

What a high priority thread on one system means can be different from what it means on another system. On one system it may mean that a high priority thread will run to completion before a lower priority thread is allowed to run. On another system, it may mean that the high priority thread may get more CPU time than a lower priority thread, but it will still run.

For this reason, threads are not as portable as other aspects of the Java language.

## Summary

- Threads are a useful technique for handling tasks that can be executed concurrently
- Threads are create created by either extending the Thread class or by implementing the Runnable interface
- Threads can be started or stopped easily
- Threads can be placed in a sleep state
- When data is shared between threads, critical sections of code must be synchronized.
- Threads can be assigned priorities