Multithreaded Programming

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each **thread defines a separate path of execution.** Thus, multithreading is a specialized form of multitasking.
There are two distinct types of multitasking: **process-based and thread-based.**

Process-based vs Thread-based multitasking

1. A *process* is a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
   In a *thread-based* multitasking environment a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

2. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
   In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code.

3. Process-based multitasking deals with the "big picture," and thread-based multitasking handles the details.

4. Multitasking threads require less overhead than multitasking processes.

5. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.
   Threads, on the other hand, share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's control.

Advantages of multithreading

- **Improved throughput**. Many concurrent compute operations and I/O requests within a single process.
- **Efficient utilization of processor (cores)** for computation and I/O
- **Superior application responsiveness**. If a request can be launched on its own thread, applications will not block or wait for the completion of pending request.

- **Improved server responsiveness**. Large or complex requests or slow clients don't block other requests for service. The overall throughput of the server is much greater.
- **Minimized system resource usage**. Threads impose minimal impact on system resources. Threads require less overhead to create, maintain, and manage than a traditional process.
- **Program structure simplification**. Threads can be used to simplify the structure of complex applications, such as multimedia applications. Simple routines can be written for each activity, making complex programs easier to design and code, and more adaptive to a wide variation in user demands.
- **Better communication**. Thread synchronization functions can be used to provide enhanced interprocess communication. In addition, sharing large amounts of data through separate threads of execution within the same address space provides extremely high-bandwidth, low-latency communication between separate tasks within an application.

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.

In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

Single-threaded systems use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Java's multithreading features work in both single and multi-core types of systems. In a single core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations

Java's traditional multithreading capabilities are described here.

Threads exist in several states. A thread can be ***running.*** It can be ***ready to run*** as soon as it gets CPU time. A running thread can be ***suspended,*** which temporarily halts its activity. A suspended thread can then be ***resumed,*** allowing it to pick up where it left off. A thread can be ***blocked*** when waiting for a resource. At any time, a thread can be **terminated**, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:

• *A thread can voluntarily relinquish control*. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

• *A thread can be preempted by a higher-priority thread*. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging
After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface. The **Thread** class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:
• It is the thread from which other "child" threads will be spawned.
• Often, it must be the last thread to finish execution because it performs various shutdown actions.
Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown here:
static Thread currentThread( )
This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

```
//example- Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
try {
for(int n = 5; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted");
}
}
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep( )** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1

Notice the output produced when **t** is used as an argument to **println( )**. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed. Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

static void sleep(long *milliseconds*) throws InterruptedException

The number of milliseconds to suspend is specified in *milliseconds*. This method may throw an **InterruptedException**. The **sleep( )** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

static void sleep(long *milliseconds*, int *nanoseconds*) throws InterruptedException

This second form is useful only in environments that allow timing periods as short as nanoseconds. As the preceding program shows, you can set the name of a thread by using **setName( )**. You can obtain the name of a thread by calling **getName( )** (but note that this is not shown in the program). These methods are members of the **Thread** class and are declared like this:
final void setName(String *threadName*)
final String getName( )
Here, *threadName* specifies the name of the thread.

<u>Creating a Thread</u>
In the most general sense, you create a thread by instantiating an object of type **Thread**.
Java defines two ways in which this can be accomplished:
• You can implement the **Runnable** interface.
• You can extend the **Thread** class, itself.

<u>Implementing Runnable Interface</u>

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns. After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb,* String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:
void start( )

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}}
class ThreadDemo {
public static void main(String args[ ] ) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object. Next, **start( )** is called, which starts the thread of execution beginning at the **run( )**

method. This causes the child thread's **for** loop to begin. After calling **start( )**, **NewThread**'s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU in single core systems, until their loops finish. The output produced by this program is as follows.

(Your output may vary based upon the specific execution environment.)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may "hang." The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
```

```
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**. Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)

Here, *threadName* specifies the name of the thread.

Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is **run( )**. This is, of course, the same method required when you implement **Runnable**. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**. Also, by implementing **Runnable**, your thread class does not need to inherit **Thread**, making it free to inherit a different class. Ultimately, which approach to use is up to you.

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
```

```
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}
}class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
```

Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main( )**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Example 1

```
package UNIT3;
/* The venue of an event has a seating capacity of 50. Tickets need to be cut and the seat number need to be shown
 * for 50 seats. * Write a multi-threaded Java program to spawn 2 threads from main to share the same run method.
 * One thread must show the message :"Cut the ticket", ticketno. Other must show the message:"Show your seat number",
 * seatno.
 */
public class MyThread extends Thread
{
//Declare a String variable to represent task.
String activity;
MyThread(String activity)
{
this.activity = activity;
}
public void run()
{
for(int i = 1; i <= 50; i++)
{
System.out.println(activity + " : " +i);
try
{
 Thread.sleep(1000); // Pause the thread execution for 1000 milliseconds.
}
catch(InterruptedException ie) {
System.out.println(ie.getMessage());
}
} // end of for loop.
```

```
} // end of run() method.
public static void main(String[] args)
{
//Create two objects to represent two tasks.
MyThread th1 = new MyThread("Cut the ticket"); // Passing task as an argument to its
constructor.
MyThread th2 = new MyThread("Show your seat number");

//Create two objects of Thread class and pass two objects as parameter to constructor of Thread
class.
Thread t1 = new Thread(th1);
Thread t2 = new Thread(th2);
t1.start();
t2.start();
}
}
```

Output sample
Show your seat number : 1
Cut the ticket : 1
Show your seat number : 2
Cut the ticket : 2
Cut the ticket : 3
Show your seat number : 3
……..

Using isAlive( ) and join( )

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )**

allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate. Here is an improved version of the preceding example that uses **join( )** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive( )** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
}
class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: " + ob1.t.isAlive());
System.out.println("Thread Two is alive: " + ob2.t.isAlive());
System.out.println("Thread Three is alive: " + ob3.t.isAlive());
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: " + ob2.t.isAlive());
System.out.println("Thread Three is alive: " + ob3.t.isAlive());
```

System.out.println("Main thread exiting.");
}
}

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.

One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

As you can see, after the calls to **join( )** return, the threads have stopped executing.

Exercise

Write a java program to spawn 2 threads from main. One thread generates odd numbers from 1 to 10 and finds sum. Other thread generates even numbers from 1 to 10 and finds sum. The main thread should wait for the threads to complete and then find total sum of odd and even numbers.

```
//Using join() to wait for threads to finish.
class oddThread implements Runnable {
String name; // name of thread
int sum=0;
Thread t;
```

```
oddThread (String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("Odd Thread: " + t);
t.start(); // Start the thread
}
//This is the entry point for thread.
public void run() {
for(int i = 1; i <=10; i=i+2) {
System.out.println(name + ": " + i);
sum+=i;
}
System.out.println("Odd sum: " + sum);
System.out.println(name + " exiting.");
}
}
class evenThread implements Runnable {
String name; // name of thread
int sum=0;
Thread t;
evenThread (String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("Even thread: " + t);
t.start(); // Start the thread
}
//This is the entry point for thread.
public void run() {
for(int i = 0; i <=10; i=i+2) {
System.out.println(name + ": " + i);
sum+=i;
}
System.out.println("Even sum: " + sum);
System.out.println(name + " exiting.");
}
}
class OddEvenThread{
public static void main(String args[]) {
        int sum=0;
oddThread ob1 = new oddThread ("Odd Thread");
evenThread ob2 = new evenThread ("Even  Thread");
try {
ob1.t.join();
ob2.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
```

```
}
System.out.println("odd-even sum="+ (ob1.sum+ob2.sum));
}
}
```

Output:

```
Odd Thread: Thread[Odd Thread,5,main]
Odd Thread: 1
Odd Thread: 3
Odd Thread: 5
Odd Thread: 7
Odd Thread: 9
Odd sum: 25
Odd Thread exiting.
Even thread: Thread[Even  Thread,5,main]
Even  Thread: 0
Even  Thread: 2
Even  Thread: 4
Even  Thread: 6
Even  Thread: 8
Even  Thread: 10
Even sum: 30
Even  Thread exiting.
odd-even sum=55
```

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run. To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**. You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

final int getPriority( )

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

Exercise

```
package UNIT3;
/*Write a java program to spawn 2 threads from main. One thread generates prime numbers
from 1 to 50 and finds sum. Other thread generates fibonacci series numbers from 1 to 50
 and finds sum. Set the priority of Prime and Fibonacci  threads to 9 and 2 respectively */
class PrimeThread implements Runnable {
String name; // name of thread
Thread t;
PrimeThread(String threadname) {
name = threadname;
t = new Thread(this, name);
t.setPriority(9);
t.start(); // Start the thread
}
//This is the entry point for thread.
public void run() {
int sum=0, flag;
for(int i = 1; i <=50; i++) {
		flag=0;
		for (int j=2;j<=i/2;j++)
    if (i%j==0) {flag=1;break;}
    if(flag==0)
    {System.out.println(name + ": " + i);
    sum+=i;}
}
System.out.println("Prime sum: " + sum);
System.out.println(name + " exiting.");
}
}
class FibThread implements Runnable {
```

```
String name; // name of thread
Thread t;
FibThread(String threadname) {
name = threadname;
t = new Thread(this, name);
t.setPriority(2);
t.start(); // Start the thread
}
//This is the entry point for thread.
public void run() {
        int sum=0;int f1=1,f2=1,f3;
        System.out.println(name + ": " + f1);
        System.out.println(name + ": " + f2);
        f3=f1+f2;
   while(f3<=500) {
     System.out.println(name + ": " + f3);
     f3=f1+f2;
     sum+=f3;
     f1=f2;f2=f3;
}
System.out.println("Fibonacci sum: " + sum);
System.out.println(name + " exiting.");
}
}
class PrimeFibThreads{
public static void main(String args[]) {
        int sum=0;
        PrimeThread ob1 = new PrimeThread("Prime Thread");
        System.out.println("Prime Thread: " + ob1.t);
   FibThread ob2 = new FibThread("Fibonacci  Thread");
   System.out.println("Fibonacci Thread: " +ob2.t);
}
}
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}

}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run() {
target.call(msg);
}
}
```

```
class Synch {
public static void main(String args[]) {
Callme target = new Callme();
new Caller(target, "Hello");
new Caller(target, "Synchronized");
new Caller(target, "World");
}
}
```

Here is the output produced by this program:

```
[Hello[Synchronized[World]
]
]
```

As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next. To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
synchronized void call(String msg) {
...
```

This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods.
Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

synchronized(*objRef*) {
// statements to be synchronized
}

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.

For example, the **run( )** method of preceding example can be as below.

// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}

Here, the **call( )** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller**'s **run( )** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Exercise

Write a java program to spawn 2 threads from main. The spawned threads will generate prime numbers from 1 to 50 by calling the method void generatePrimenumbers() of another class namely generatePrime. Prevent the 2 threads from racing to complete generatePrimenumbers() by declaring it as synchronized.

```
import java.lang.*;
class generatePrime {

	synchronized void generateprimenumber()
	{int i,j,l=0;
		for(i=2;i<50;i++)
		{
			for(j=2;j<i/2;j++)
			{
				if(i%j==0)
				{
					l++;
				}
			}
			if(l==0)
			{
				System.out.print(i+"\t");
			}
			l=0;
		}
		System.out.println();
	}
	}
class primes implements Runnable
{
```

```
	Thread t;
	generatePrime obj;
	primes(generatePrime  ob)
	{
		t=new Thread(this);
		obj=ob;
		t.start();
	}
	public void run()
	{
		obj.generateprimenumber();
	}
}
public class thread {
	public static void main(String args[])
	{
		generatePrime obj=new generatePrime ();
		primes ob1=new primes(obj);
		primes ob2=new primes(obj);
	}
}
```

Output:

```
2	3	4	5	7	11	13	17	19	23	29	31	37
	41	43	47
2	3	4	5	7	11	13	17	19	23	29	31	37
	41	43	47
```

Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more precise level of control through interthread communication.

To avoid polling, Java includes an elegant interthread communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context. The rules for using these methods are actually quite simple:

• **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( ).
• **notify( )** wakes up a thread that called wait( ) on the same object.
• **notifyAll( )** wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

These methods are declared within Object, as shown here:
final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )

Additional forms of wait( ) exist that allow you to specify a period of time to wait. Before working through an example that illustrates interthread communication, an important point needs to be made. Although wait( ) normally waits until notify( ) or notifyAll( ) is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a spurious wakeup. In this case, a waiting thread resumes without notify( ) or notifyAll( ) having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, Oracle recommends that calls to wait( ) should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Let's now work through an example that uses wait( ) and notify( ). To begin, consider the following sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: Q, the queue that you're trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

```
// An incorrect implementation of a producer and consumer.
class Q {
int n;
synchronized int get() {
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n) {
this.n = n;
System.out.println("Put: " + n);
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
```

```
public void run() {
while(true) {
q.get();
}
}
}
class PC {
public static void main(String args[]) {

Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Although the put( ) and get( ) methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use wait( ) and notify( ) to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
while(!valueSet)
try {
wait();
```

```
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {

while(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
```

```
}
class PCFixed {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Inside get( ), wait( ) is called. This causes its execution to suspend until Producer notifies you that some data is ready. When this happens, execution inside get( ) resumes. After the data has been obtained, get( ) calls notify( ). This tells Producer that it is okay to put more data in the queue. Inside put( ), wait( ) suspends execution until Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and notify( ) is called. This tells Consumer that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

Deadlock is a situation where two threads have a circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

• In general, it occurs only rarely, when the two threads time-slice in just the right way.
• It may involve more than two threads and two synchronized objects.

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, A and B, with methods foo( ) and bar( ), respectively, which pause briefly before trying to call a method in the other class. The main class, named Deadlock, creates an A and a B instance, and then starts a second thread to set up the deadlock condition. The foo( ) and bar( ) methods use sleep( ) as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A {
synchronized void foo(B b) {
String name = Thread.currentThread().getName();
System.out.println(name + " entered A.foo");
try {
Thread.sleep(1000);
} catch(Exception e) {
System.out.println("A Interrupted");

}
System.out.println(name + " trying to call B.last()");
b.last();
}
synchronized void last() {
System.out.println("Inside A.last");
}
}
class B {
synchronized void bar(A a) {
String name = Thread.currentThread().getName();
System.out.println(name + " entered B.bar");
try {
Thread.sleep(1000);
} catch(Exception e) {
System.out.println("B Interrupted");
}
System.out.println(name + " trying to call A.last()");
a.last();
}
synchronized void last() {
System.out.println("Inside A.last");
}
}
class Deadlock implements Runnable {
A a = new A();
B b = new B();
Deadlock() {
Thread.currentThread().setName("MainThread");
Thread t = new Thread(this, "RacingThread");
t.start();
a.foo(b); // get lock on a in this thread.
System.out.println("Back in main thread");
}
public void run() {
b.bar(a); // get lock on b in other thread.
```

```
System.out.println("Back in other thread");
}
public static void main(String args[]) {
new Deadlock();
}
}
```

Output

```
RacingThread entered B.bar
MainThread entered A.foo
MainThread trying to call B.last()
RacingThread trying to call A.last()
```