

Chapter 12

Multithreaded Programming

Multithreading Fundamentals

There are two distinct types of multitasking: process-based and thread-based.

Process-based multitasking is the feature that allows your computer to run two or more programs concurrently.

For example, it is process-based multitasking that allows you to run the Java compiler at the same time you are using a text editor or browsing the Internet. 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. This means that a single program can perform two or more tasks at once. For instance, a text editor can be formatting text while it is printing, as long as these two actions are being performed by two separate threads.

Multithreading Fundamentals...

A principal advantage of multithreading is that it enables you to write very efficient programs because it lets you utilize the idle time that is present in most programs.

As you probably know, most I/O devices, whether they be network ports, disk drives, or the keyboard, are much slower than the CPU. Thus, a program will often spend a majority of its execution time waiting to send or receive information to or from a device.

By using multithreading, your program can execute another task during this idle time. For example, while one part of your program is sending a file over the Internet, another part can be reading keyboard input, and still another can be buffering the next block of data to send.

Multithreading Fundamentals...

Java's multithreading features work in two 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 **multiprocessor/multicore 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.

Multithreading Fundamentals...

A thread can be in one of several states. It can be ***running***. It can be *ready to run* as soon as it gets CPU time. A running thread can be ***suspended***, which is a temporary halt to its execution. It can later be ***resumed***. A thread can be ***blocked*** when waiting for a resource. A thread can be ***terminated***, in which case its execution ends and cannot be resumed.


Along with thread-based multitasking comes the need for a special type of feature called ***synchronization***, which allows the execution of threads to be coordinated in certain well-defined ways. Java has a complete subsystem devoted to synchronization

The Thread Class and Runnable Interface

Java's multithreading system is built upon the **Thread** class and its companion interface, **Runnable**. Both are packaged in **java.lang**. **Thread** encapsulates a thread of execution. 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. Here are some of the more commonly used ones (we will be looking at these more closely as they are used):

Method	Meaning
<code>final String getName()</code>	Obtains a thread's name.
<code>final int getPriority()</code>	Obtains a thread's priority.
<code>final boolean isAlive()</code>	Determines whether a thread is still running.
<code>final void join()</code>	Waits for a thread to terminate.
<code>void run()</code>	Entry point for the thread.
<code>static void sleep(long <i>milliseconds</i>)</code>	Suspends a thread for a specified period of milliseconds.
<code>void start()</code>	Starts a thread by calling its run() method.

All processes have at least one thread of execution, which is usually called the *main thread*, because it is the one that is executed when your program begins. Thus, the main thread is the thread that all of the preceding example programs  have been using. From the main thread, you can create other threads.

Creating a Thread

You create a thread by instantiating an object of type **Thread**. The **Thread** class encapsulates an object that is runnable. As mentioned, Java defines two ways in which you can create a runnable object:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class.

The **Runnable** interface abstracts a unit of executable code. You can construct a thread on any object that implements the **Runnable** interface. **Runnable** defines only one 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. 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 have created a class that implements **Runnable**, you will instantiate an object of type **Thread** on an object of that class. **Thread** defines several constructors. The one that we will use first is shown here:

```
Thread(Runnable threadOb)
```


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

Once created, the new thread 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 thread by implementing Runnable.
```

```
class MyThread implements Runnable {  
    String thrdName;
```

Objects of **MyThread** can be run in their own threads because **MyThread** implements **Runnable**.

```
MyThread(String name) {  
    thrdName = name;  
}  
  
// Entry point of thread.  
public void run() { ←———— Threads start executing here.  
    System.out.println(thrdName + " starting.");  
    try {  
        for(int count=0; count < 10; count++) {  
            Thread.sleep(400);  
            System.out.println("In " + thrdName +  
                               ", count is " + count);  
        }  
    }  
    catch(InterruptedException exc) {  
        System.out.println(thrdName + " interrupted.");  
    }  
    System.out.println(thrdName + " terminating.");  
}
```

```
class UseThreads {
    public static void main(String args[]) {
        System.out.println("Main thread starting.");

        // First, construct a MyThread object.
        MyThread mt = new MyThread("Child #1"); ← Create a runnable object.

        // Next, construct a thread from that object.
        Thread newThrd = new Thread(mt); ← Construct a thread on that object.

        // Finally, start execution of the thread.
        newThrd.start(); ← Start running the thread.

        for(int i=0; i<50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        }


        System.out.println("Main thread ending.");
    }
}
```

```
Main thread starting.  
.Child #1 starting.  
...In Child #1, count is 0  
....In Child #1, count is 1  
....In Child #1, count is 2  
...In Child #1, count is 3  
....In Child #1, count is 4  
....In Child #1, count is 5  
....In Child #1, count is 6  
...In Child #1, count is 7  
....In Child #1, count is 8  
....In Child #1, count is 9  
Child #1 terminating.  
.....Main thread ending.
```

One other point: In a multithreaded program, you often will want the main thread to be the last thread to finish running. As a general rule, a program continues to run until all of its threads have ended. Thus, having the main thread finish last is not a requirement. It is, however, often a good practice to follow—especially when you are first learning about threads.

```
// Improved MyThread.
```


```
class MyThread implements Runnable {
```

```
    Thread thrd;  A reference to the thread is stored in thrd.
```

```
    // Construct a new thread.
```

```
    MyThread(String name) {
```

```
        thrd = new Thread(this, name);  The thread is named when it is created.
```

```
        thrd.start(); // start the thread  Begin executing the thread.
```

```
    }
```

```
    // Begin execution of new thread.
```

```
    public void run() {
```

```
        System.out.println(thrd.getName() + " starting.");
```

```
        try {
```

```
            for(int count=0; count<10; count++) {
```

```
                Thread.sleep(400);
```

```
                System.out.println("In " + thrd.getName() +
```

```
                    ", count is " + count);
```

```
            }
```

```
        }
```

```
        catch (InterruptedException exc) {  
            System.out.println(thrd.getName() + " interrupted.");  
        }  
        System.out.println(thrd.getName() + " terminating.");  
    }  
}
```

```
class UseThreadsImproved {  
    public static void main(String args[]) {  
        System.out.println("Main thread starting.");  
  
        MyThread mt = new MyThread("Child #1");  
  
        for(int i=0; i < 50; i++) {  
            System.out.print(".");  
            try {  
                Thread.sleep(100);  
            }  
            catch (InterruptedException exc) {  
                System.out.println("Main thread interrupted.");  
            }  
        }  
  
        System.out.println("Main thread ending.");  
    }  
}
```

↑ Now the thread starts when it is created.

Extending Thread

Implementing **Runnable** is one way to create a class that can instantiate thread objects. Extending **Thread** is the other. In this project, you will see how to extend **Thread** by creating a program functionally identical to the **UseThreadsImproved** program.

When a class extends **Thread**, it 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. It is possible to override other **Thread** methods, but doing so is not required.

```
Extend Thread.  
*/  
class MyThread extends Thread {  
  
    // Construct a new thread.  
    MyThread(String name) {  
        super(name); // name thread  
        start(); // start the thread  
    }  
  
    // Begin execution of new thread.
```

```
// Begin execution of new thread.
public void run() {
    System.out.println(getName() + " starting.");
    try {
        for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("In " + getName() +
                               ", count is " + count);
        }
    }
    catch(InterruptedException exc) {
        System.out.println(getName() + " interrupted.");
    }

    System.out.println(getName() + " terminating.");
}
}
```



```
class ExtendThread {
    public static void main(String args[]) {
        System.out.println("Main thread starting.");

        MyThread mt = new MyThread("Child #1");

        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException exc) {
                System.out.println("Main thread interrupted.");
            }
        }

        System.out.println("Main thread ending.");
    }
}
```

```
// Create multiple threads.

class MyThread implements Runnable {
    Thread thrd;

    // Construct a new thread.
    MyThread(String name) {
        thrd = new Thread(this, name);

        thrd.start(); // start the thread
    }

    // Begin execution of new thread.
    public void run() {

        System.out.println(thrd.getName() + " starting.");
    }
}
```

```
try {
    for(int count=0; count < 10; count++) {
        Thread.sleep(400);
        System.out.println("In " + thrd.getName() +
                           ", count is " + count);
    }
}
catch(InterruptedException exc) {
    System.out.println(thrd.getName() + " interrupted.");
}
System.out.println(thrd.getName() + " terminating.");
}
```

```
class MoreThreads {
    public static void main(String args[]) {
        System.out.println("Main thread starting.");
    }
}
```

```
MyThread mt1 = new MyThread("Child #1");
```

```
MyThread mt2 = new MyThread("Child #2");
```

```
MyThread mt3 = new MyThread("Child #3");
```

← Create and start
executing three threads.

```
for(int i=0; i < 50; i++) {
```

```
    System.out.print(".");
```

```
    try {
```

```
        Thread.sleep(100);
```

```
    }
```

```
    catch(InterruptedException exc) {
```

```
        System.out.println("Main thread interrupted.");
```

```
    }
```

```
}
```

```
System.out.println("Main thread ending.");
```

```
}
```

```
}
```

Main thread starting.
Child #1 starting.
.Child #2 starting.
Child #3 starting.
...In Child #3, count is 0
In Child #2, count is 0
In Child #1, count is 0
....In Child #1, count is 1
In Child #2, count is 1
In Child #3, count is 1
....In Child #2, count is 2
In Child #3, count is 2
In Child #1, count is 2
...In Child #1, count is 3
In Child #2, count is 3
In Child #3, count is 3
....In Child #1, count is 4
In Child #3, count is 4
In Child #2, count is 4

....In Child #1, count is 5
In Child #3, count is 5
In Child #2, count is 5
...In Child #3, count is 6
.In Child #2, count is 6
In Child #1, count is 6
...In Child #3, count is 7
In Child #1, count is 7
In Child #2, count is 7
....In Child #2, count is 8
In Child #1, count is 8
In Child #3, count is 8
....In Child #1, count is 9
Child #1 terminating.
In Child #2, count is 9
Child #2 terminating.
In Child #3, count is 9
Child #3 terminating.
.....Main thread ending.

Determining When a Thread Ends

Fortunately, **Thread** provides two means by which you can determine if a thread has ended. First, you can call **isAlive()** on the thread. 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. To try **isAlive()**, substitute this version of **MoreThreads** for the one shown in the preceding program:

```
// Use isAlive().
class MoreThreads {
    public static void main(String args[]) {
        System.out.println("Main thread starting.");

        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");
```

```
do {  
    System.out.print(".");  
    try {  
        Thread.sleep(100);  
    }  
    catch (InterruptedException exc) {  
        System.out.println("Main thread interrupted.");  
    }  
} while (mt1.thrd.isAlive() ||  
        mt2.thrd.isAlive() || ← This waits until all threads terminate.  
        mt3.thrd.isAlive());
```

```
System.out.println("Main thread ending.");
```

```
}
```

```
}
```

This version produces output that is similar to the previous version, except that **main()** ends as soon as the other threads finish. The difference is that it uses **isAlive()** to wait for the child threads to terminate. Another way to wait for a thread to finish is to call **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 a program that uses **join()** to ensure that the main thread is the last to stop:

```
// Use join().
```

```
class MyThread implements Runnable {  
    Thread thrd;
```



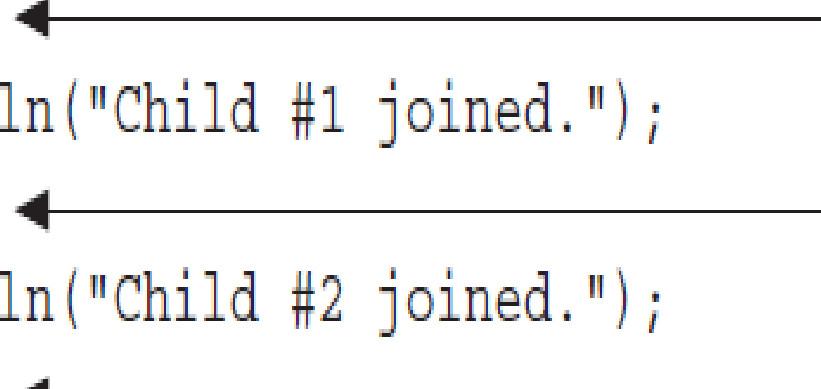
```
// Construct a new thread.
MyThread(String name) {
    thrd = new Thread(this, name);
    thrd.start(); // start the thread
}

// Begin execution of new thread.
public void run() {
    System.out.println(thrd.getName() + " starting.");
    try {
        for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("In " + thrd.getName() +
                               ", count is " + count);
        }
    }
}
```

```
        catch (InterruptedException exc) {  
            System.out.println(thrd.getName() + " interrupted.");  
        }  
        System.out.println(thrd.getName() + " terminating.");  
    }  
}
```

```
class JoinThreads {  
    public static void main(String args[]) {  
        System.out.println("Main thread starting.");  
  
        MyThread mt1 = new MyThread("Child #1");  
        MyThread mt2 = new MyThread("Child #2");  
        MyThread mt3 = new MyThread("Child #3");
```

```
try {  
    mt1.thrd.join();  
    System.out.println("Child #1 joined.");  
    mt2.thrd.join();  
    System.out.println("Child #2 joined.");  
    mt3.thrd.join();  
    System.out.println("Child #3 joined.");  
}  
catch (InterruptedException exc) {  
    System.out.println("Main thread interrupted.");  
}  
    System.out.println("Main thread ending.");  
}
```



Wait until the specified thread ends.

Main thread starting.
Child #1 starting.
Child #2 starting.
Child #3 starting.
In Child #2, count is 0
In Child #1, count is 0
In Child #3, count is 0
In Child #2, count is 1
In Child #3, count is 1
In Child #1, count is 1
In Child #2, count is 2
In Child #1, count is 2
In Child #3, count is 2
In Child #2, count is 3
In Child #3, count is 3
In Child #1, count is 3
In Child #3, count is 4
In Child #2, count is 4
In Child #1, count is 4
In Child #3, count is 5
In Child #1, count is 5

In Child #2, count is 5
In Child #3, count is 6
In Child #2, count is 6
In Child #1, count is 6
In Child #3, count is 7
In Child #1, count is 7
In Child #2, count is 7
In Child #3, count is 8
In Child #2, count is 8
In Child #1, count is 8
In Child #3, count is 9
Child #3 terminating.
In Child #2, count is 9
Child #2 terminating.
In Child #1, count is 9
Child #1 terminating.
Child #1 joined.
Child #2 joined.
Child #3 joined.
Main thread ending.

Thread Priorities

Each thread has associated with it a priority setting. A thread's priority determines, in part, how much CPU time a thread receives relative to the other active threads. In general, over a given period of time, low-priority threads receive little. High-priority threads receive a lot. As you might expect, how much CPU time a thread receives has profound impact on its execution characteristics and its interaction with other threads currently executing in the system.

It is important to understand that factors other than a thread's priority also affect how much CPU time a thread receives. For example, if a high-priority thread is waiting on some resource, perhaps for keyboard input, then it will be blocked, and a lower priority thread will run. However, when that high-priority thread gains access to the resource, it can preempt the low-priority thread and resume execution. Another factor that affects the scheduling of threads is the way the operating system implements multitasking. (See "Ask the Expert" at the end of this section.) Thus, just because you give one thread a high priority and another a low priority does not necessarily mean that one thread will run faster or more often than the other. It's just that the high-priority thread has greater potential access to the CPU.

When a child thread is started, its priority setting is equal to that of its parent thread. You can change a thread's priority by calling **setPriority()**, 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( )
```

Synchronization

When using multiple threads, it is sometimes necessary to coordinate the activities of two or more. The process by which this is achieved is called *synchronization*. The most common reason for synchronization is when two or more threads need access to a shared resource that can be used by only one thread at a time. For example, when one thread is writing to a file, a second thread must be prevented from doing so at the same time. Another reason for synchronization is when one thread is waiting for an event that is caused by another thread. In this case, there must be some means by which the first thread is held in a suspended state until the event has occurred. Then, the waiting thread must resume execution.

Key to synchronization in Java is the concept of the *monitor*, which controls access to an object. A monitor works by implementing the concept of a *lock*. When an object is locked by one thread, no other thread can gain access to the object. When the thread exits, the object is unlocked and is available for use by another thread.

All objects in Java have a monitor. This feature is built into the Java language, itself. Thus, all objects can be synchronized. Synchronization is supported by the keyword **synchronized** and a few well-defined methods that all objects have. Since synchronization was designed into Java from the start, it is much easier to use

There are two ways that you can synchronize your code. Both involve the use of the **synchronized** keyword

Using Synchronized Methods

You can synchronize access to a method by modifying it with the **synchronized** keyword. When that method is called, the calling thread enters the object's monitor, which then locks the object. While locked, no other thread can enter the method, or enter any other synchronized method defined by the object's class. When the thread returns from the method, the monitor unlocks the object, allowing it to be used by the next thread. Thus, synchronization is achieved with virtually no programming effort on your part.


```
// Use synchronize to control access.
```

```
class SumArray {  
    private int sum;
```

```
    synchronized int sumArray(int nums[]) { ←———— sumArray() is synchronized.  
        sum = 0; // reset sum
```

```
        for(int i=0; i<nums.length; i++) {  
            sum += nums[i];  
            System.out.println("Running total for " +  
                               Thread.currentThread().getName() +  
                               " is " + sum);
```

```
        try {  
            Thread.sleep(10); // allow task-switch  
        }
```

```
        catch (InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }
    return sum;
}
}
```

```
class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int a[];
    int answer;

    // Construct a new thread.
    MyThread(String name, int nums[]) {
        thrd = new Thread(this, name);
    }
}
```

```
        a = nums;
        thrd.start(); // start the thread
    }

    // Begin execution of new thread.
    public void run() {
        int sum;

        System.out.println(thrd.getName() + " starting.");

        answer = sa.sumArray(a);
        System.out.println("Sum for " + thrd.getName() +
                           " is " + answer);

        System.out.println(thrd.getName() + " terminating.");
    }
}
```

```
class Sync {  
    public static void main(String args[]) {  
        int a[] = {1, 2, 3, 4, 5};  
  
        MyThread mt1 = new MyThread("Child #1", a);  
        MyThread mt2 = new MyThread("Child #2", a);  
  
        try {  
            mt1.thrd.join();  
            mt2.thrd.join();  
        }  
        catch (InterruptedException exc) {  
            System.out.println("Main thread interrupted.");  
        }  
    }  
}
```

Child #1 starting.
Running total for Child #1 is 1
Child #2 starting.
Running total for Child #1 is 3
Running total for Child #1 is 6
Running total for Child #1 is 10
Running total for Child #1 is 15
Sum for Child #1 is 15
Child #1 terminating.
Running total for Child #2 is 1
Running total for Child #2 is 3
Running total for Child #2 is 6
Running total for Child #2 is 10
Running total for Child #2 is 15
Sum for Child #2 is 15
Child #2 terminating.

- A synchronized method is created by preceding its declaration with **synchronized**.
- For any given object, once a synchronized method has been called, the object is locked and no synchronized methods on the same object can be used by another thread of execution.
- Other threads trying to call an in-use synchronized object will enter a wait state until the object is unlocked.
- When a thread leaves the synchronized method, the object is unlocked.

MULTITHREADING - 2

The `synchronized` Statement

Although creating `synchronized` methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. For example, you might want to synchronize access to some method that is not modified by `synchronized`. This can occur because you want to use a class that was not created by you but by a third party, and you do not have access to the source code. Thus, it is not possible for you to 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 a `synchronized` block:

```
synchronized(objref) {  
    // statements to be synchronized  
}
```



```
// Use a synchronized block to control access to SumArray.
class SumArray {
    private int sum;

    int sumArray(int nums[]) { ←———— Here, sumArray()
        sum = 0; // reset sum           is not synchronized.

        for(int i=0; i<nums.length; i++) {
            sum += nums[i];
            System.out.println("Running total for " +
                               Thread.currentThread().getName() +
                               " is " + sum);
            try {
                Thread.sleep(10); // allow task-switch
            }
            catch(InterruptedException exc) {
                System.out.println("Thread interrupted.");
            }
        }
    }
}
```

```
    }  
    return sum;  
  }  
}
```

```
class MyThread implements Runnable {  
    Thread thrd;  
    static SumArray sa = new SumArray();  
    int a[];  
    int answer;  
  
    // Construct a new thread.  
    MyThread(String name, int nums[]) {  
        thrd = new Thread(this, name);  
        a = nums;  
        thrd.start(); // start the thread  
    }  
  
    // Begin execution of new thread.  
    public void run() {  
        int sum;
```

```

System.out.println(thrd.getName() + " starting.");

// synchronize calls to sumArray()
synchronized(sa) { ←————— Here, calls to sumArray()
    answer = sa.sumArray(a);      on sa are synchronized.
}
System.out.println("Sum for " + thrd.getName() +
                  " is " + answer);

    System.out.println(thrd.getName() + " terminating.");
}
}

class Sync {
    public static void main(String args[]) {
        int a[] = {1, 2, 3, 4, 5};
    }
}

```

```
MyThread mt1 = new MyThread("Child #1", a);  
MyThread mt2 = new MyThread("Child #2", a);  
  
try {  
    mt1.thrd.join();  
    mt2.thrd.join();  
} catch (InterruptedException exc) {  
    System.out.println("Main thread interrupted.");  
}  
}
```

This version produces the same, correct output as the one shown earlier that uses a synchronized method.

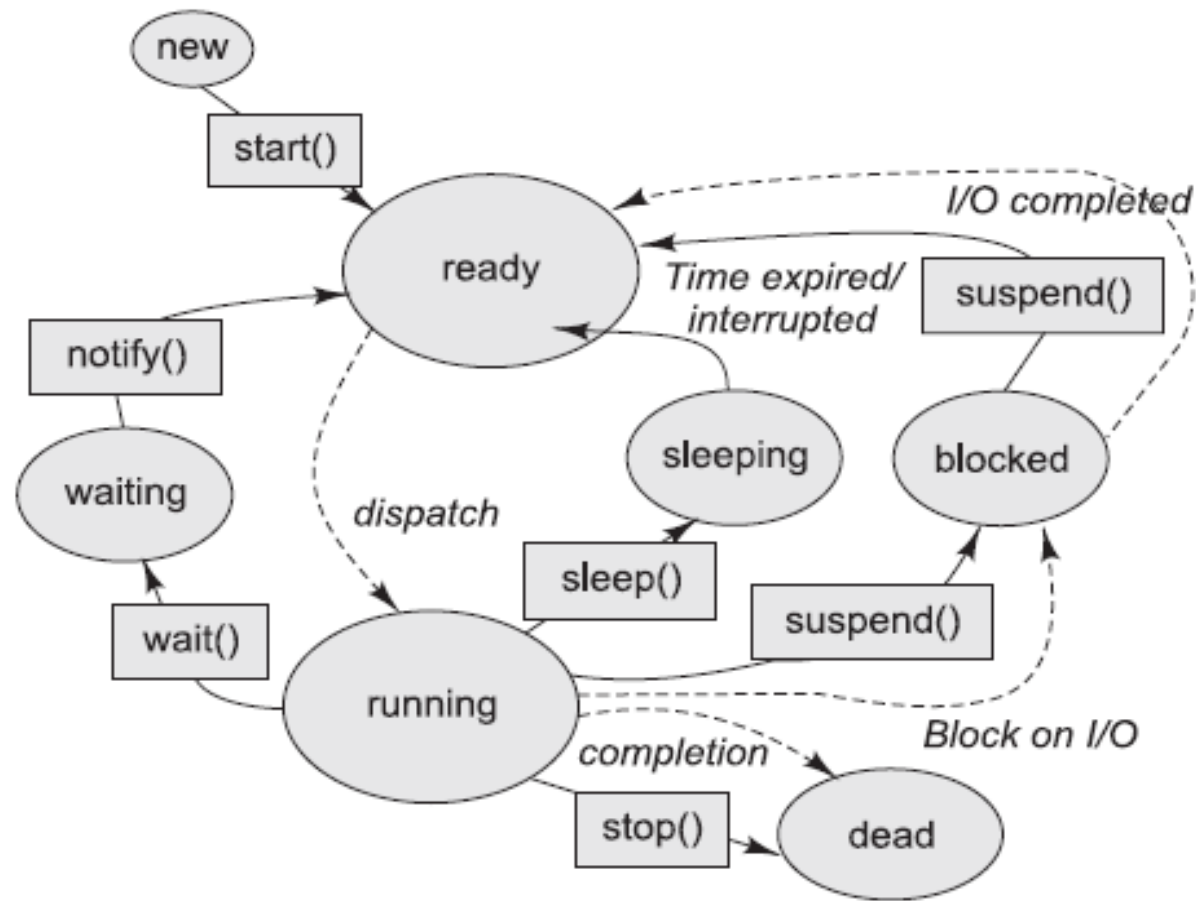


Fig. 14.4 Life cycle of Java threads

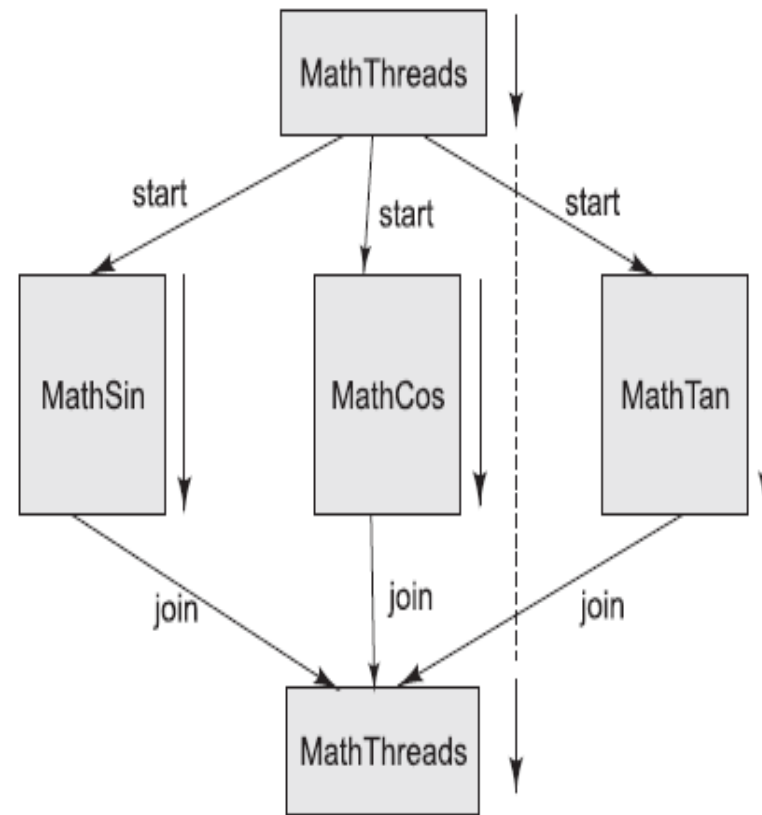


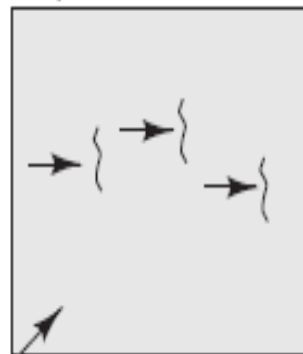
Fig. 14.5 Flow of control in a master and multiple workers threads application

Single-threaded Process



Threads of
Execution

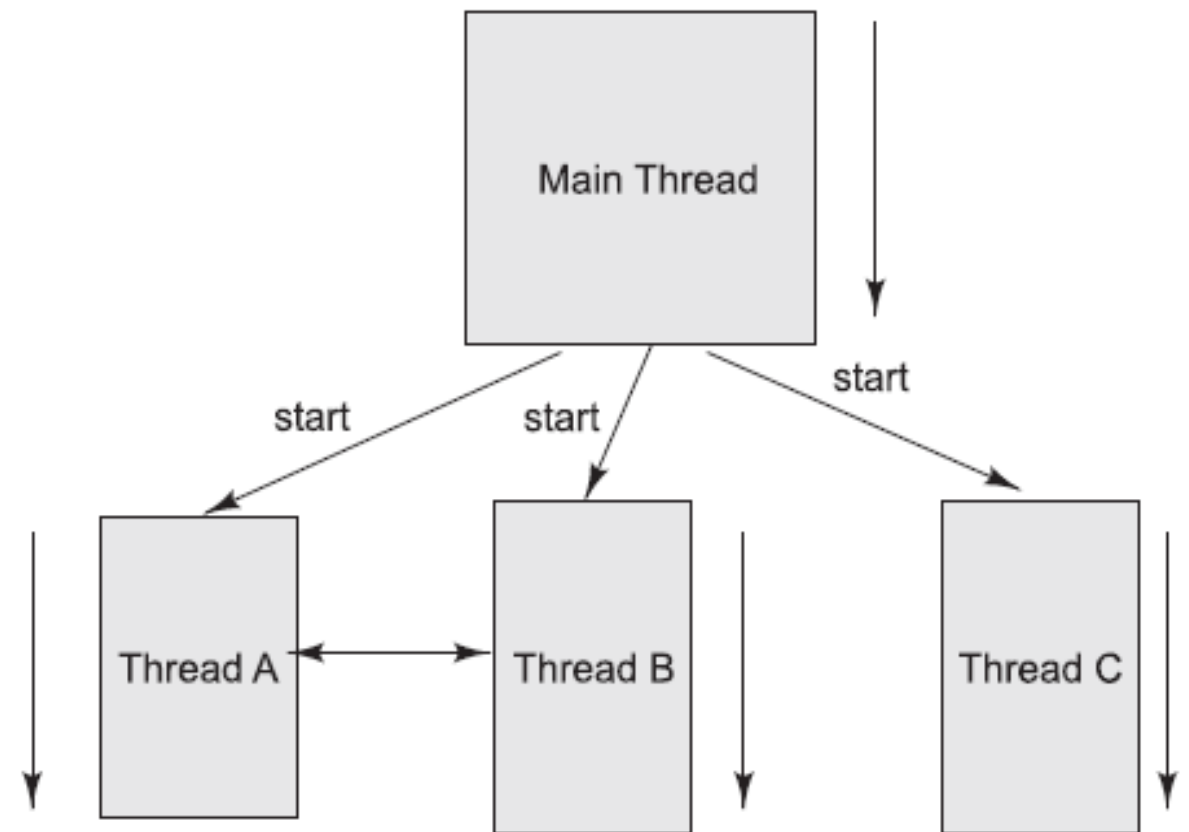
Multipletreaded Process



Single instruction stream

Common

Multiple instruction stream



Threads may switch or exchange data/results

Thread Communication Using `notify()`, `wait()`, and `notifyAll()`

Consider the following situation. A thread called T is executing inside a synchronized method and needs access to a resource called R that is temporarily unavailable. What should T do? If T enters some form of polling loop that waits for R, T ties up the object, preventing other threads' access to it. This is a less than optimal solution because it partially defeats the advantages of programming for a multithreaded environment. A better solution is to have T temporarily relinquish control of the object, allowing another thread to run. When R becomes available, T can be notified and resume execution. Such an approach relies upon some form of interthread communication in which one thread can notify another that it is blocked and be notified that it can resume execution. Java supports interthread communication with the **`wait()`**, **`notify()`**, and **`notifyAll()`** methods.

The **wait()**, **notify()**, and **notifyAll()** methods are part of all objects because they are implemented by the **Object** class. These methods should be called only from within a **synchronized** context. Here is how they are used. When a thread is temporarily blocked from running, it calls **wait()**. This causes the thread to go to sleep and the monitor for that object to be released, allowing another thread to use the object. At a later point, the sleeping thread is awakened when some other thread enters the same monitor and calls **notify()**, or **notifyAll()**.

Following are the various forms of **wait()** defined by **Object**:

`final void wait() throws InterruptedException`

`final void wait(long millis) throws InterruptedException`

`final void wait(long millis, int nanos) throws InterruptedException`

The first form waits until notified. The second form waits until notified or until the specified period of milliseconds has expired. The third form allows you to specify the wait period in terms of nanoseconds.

Here are the general forms for **notify()** and **notifyAll()**:

```
final void notify( )
```

```
final void notifyAll( )
```

A call to **notify()** resumes one waiting thread. A call to **notifyAll()** notifies all threads, with the highest priority thread gaining access to the object.

Before looking at an example that uses **wait()**, 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*.

```
// 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.");
        }
    }
```

An Example That Uses `wait()` and `notify()`

To understand the need for and the application of `wait()` and `notify()`, we will create a program that simulates the ticking of a clock by displaying the words Tick and Tock on the screen. To accomplish this, we will create a class called **TickTock** that contains two methods: `tick()` and `tock()`. The `tick()` method displays the word "Tick", and `tock()` displays "Tock". To run the clock, two threads are created, one that calls `tick()` and one that calls `tock()`. The goal is to make the two threads execute in a way that the output from the program displays a consistent "Tick Tock"—that is, a repeated pattern of one tick followed by one tock.

```
// Use wait() and notify() to create a ticking clock.
```

```
class TickTock {
```

```
    String state; // contains the state of the clock
```

```
synchronized void tick(boolean running) {  
    if(!running) { // stop the clock  
        state = "ticked";  
        notify(); // notify any waiting threads  
        return;  
    }  
  
    System.out.print("Tick ");  
  
    state = "ticked"; // set the current state to ticked  
  
    notify(); // let tock() run ←——— tick() notifies tock().  
    try {  
        while(!state.equals("tocked"))  
            wait(); // wait for tock() to complete ←——— tick() waits for tock()  
    }  
    catch(InterruptedException exc) {  
        System.out.println("Thread interrupted.");  
    }  
}
```



```
        catch(InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }

    synchronized void tock(boolean running) {
        if(!running) { // stop the clock
            state = "tocked";
            notify(); // notify any waiting threads
            return;
        }

        System.out.println("Tock");

        state = "tocked"; // set the current state to tocked

        notify(); // let tick() run ←———— tock( ) notifies tick( ).
```

```
try {
    while(!state.equals("ticked"))
        wait(); // wait for tick to complete ← tock() waits for tick().
}
catch(InterruptedException exc) {
    System.out.println("Thread interrupted.");
}
}
```

```
class MyThread implements Runnable {
    Thread thrd;
    TickTock ttOb;

    // Construct a new thread.
    MyThread(String name, TickTock tt) {
        thrd = new Thread(this, name);
        ttOb = tt;
        thrd.start(); // start the thread
    }
}
```

```
// Begin execution of new thread.
public void run() {

    if(thrd.getName().compareTo("Tick") == 0) {
        for(int i=0; i<5; i++) ttOb.tick(true);
        ttOb.tick(false);
    }
    else {
        for(int i=0; i<5; i++) ttOb.tock(true);
        ttOb.tock(false);
    }
}

}

class ThreadCom {
    public static void main(String args[]) {
        TickTock tt = new TickTock();
        MyThread mt1 = new MyThread("Tick", tt);
        MyThread mt2 = new MyThread("Tock", tt);
    }
}
```

```
try {  
    mt1.thrd.join();  
    mt2.thrd.join();  
} catch (InterruptedException exc) {  
    System.out.println("Main thread interrupted.");  
}  
}  
}
```

Here is the output produced by the program:

```
Tick Tock  
Tick Tock  
Tick Tock  
Tick Tock  
Tick Tock
```

Suspending, Resuming, and Stopping Threads

It is sometimes useful to suspend execution of a thread. For example, a separate thread can be used to display the time of day. If the user does not desire a clock, then its thread can be suspended. Whatever the case, it is a simple matter to suspend a thread. Once suspended, it is also a simple matter to restart the thread.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java and more modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend()**, **resume()**, and **stop()**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. They have the following forms:

```
final void resume( )
```

```
final void suspend( )
```

```
final void stop( )
```

While these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must no longer be used. Here's why. The **suspend()** method of the **Thread** class was deprecated by Java 2. This was done because **suspend()** can sometimes cause serious problems that involve deadlock. The **resume()** method is also deprecated. It does not cause problems but cannot be used without the **suspend()** method as its counterpart. The **stop()** method of the **Thread** class was also deprecated by Java 2. This was done because this method too can sometimes cause serious problems.

Since you cannot now use the **suspend()**, **resume()**, or **stop()** methods to control a thread, you might at first be thinking that there is no way to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run()** method periodically checks to determine if that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing two flag variables: one for suspend and resume, and one for stop. For suspend and resume, as long as the flag is set to "running," the **run()** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. For the stop flag, if it is set to "stop," the thread must terminate.

```
// Suspending, resuming, and stopping a thread

class MyThread implements Runnable {
    Thread thrd;

    boolean suspended; ←———— Suspend thread when true.
    boolean stopped; ←———— Stop thread when true.

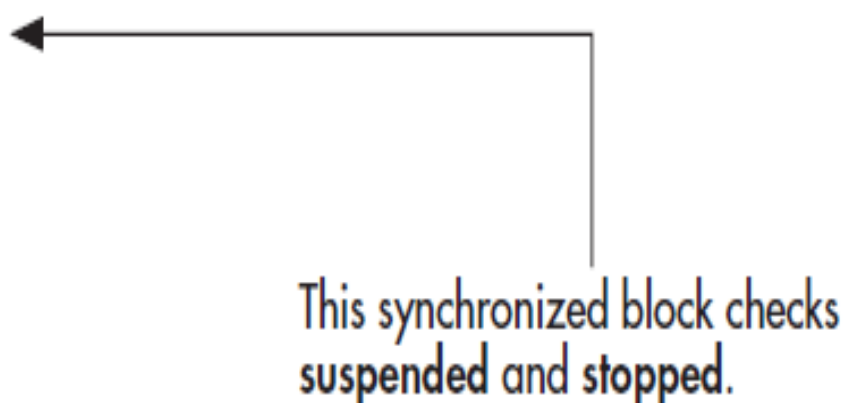
    MyThread(String name) {
        thrd = new Thread(this, name);
        suspended = false;
        stopped = false;
        thrd.start();
    }

    // This is the entry point for thread.
```



```
public void run() {
    System.out.println(thrd.getName() + " starting.");
    try {
        for(int i = 1; i < 1000; i++) {
            System.out.print(i + " ");
            if((i%10)==0) {
                System.out.println();
                Thread.sleep(250);
            }

            // Use synchronized block to check suspended and stopped.
            synchronized(this) {
                while(suspended) {
                    wait();
                }
                if(stopped) break;
            }
        }
    }
}
```



This synchronized block checks
suspended and stopped.


```
        }  
    }  
    } catch (InterruptedException exc) {  
        System.out.println(thrd.getName() + " interrupted.");  
    }  
    System.out.println(thrd.getName() + " exiting.");  
}
```

// Stop the thread.

```
synchronized void mystop() {  
    stopped = true;
```

// The following ensures that a suspended thread can be stopped.

```
suspended = false;  
notify();  
}
```

```
// Suspend the thread.
synchronized void mysuspend() {
    suspended = true;
}

// Resume the thread.
synchronized void myresume() {
    suspended = false;
    notify();
}
}

class Suspend {
    public static void main(String args[]) {
        MyThread ob1 = new MyThread("My Thread");
```

```
try {  
    Thread.sleep(1000); // let ob1 thread start executing  
  
    ob1.mysuspend();  
    System.out.println("Suspending thread.");  
    Thread.sleep(1000);  
  
    ob1.myresume();  
    System.out.println("Resuming thread.");  
    Thread.sleep(1000);  
  
    ob1.mysuspend();  
    System.out.println("Suspending thread.");  
    Thread.sleep(1000);  
}
```

```
    ob1.myresume();  
    System.out.println("Resuming thread.");  
    Thread.sleep(1000);  
  
    ob1.mysuspend();  
    System.out.println("Stopping thread.");  
    ob1.mystop();  
} catch (InterruptedException e) {  
    System.out.println("Main thread Interrupted");  
}  
  
// wait for thread to finish  
try {  
    ob1.thrd.join();  
} catch (InterruptedException e) {  
    System.out.println("Main thread Interrupted");  
}  
  
System.out.println("Main thread exiting.");
```

```
My Thread starting.  
1 2 3 4 5 6 7 8 9 10  
11 12 13 14 15 16 17 18 19 20  
21 22 23 24 25 26 27 28 29 30  
31 32 33 34 35 36 37 38 39 40  
Suspending thread.  
Resuming thread.  
41 42 43 44 45 46 47 48 49 50  
51 52 53 54 55 56 57 58 59 60  
61 62 63 64 65 66 67 68 69 70  
71 72 73 74 75 76 77 78 79 80  
Suspending thread.  
Resuming thread.  
81 82 83 84 85 86 87 88 89 90  
91 92 93 94 95 96 97 98 99 100  
101 102 103 104 105 106 107 108 109 110  
111 112| 113 114 115 116 117 118 119 120  
Stopping thread.  
My Thread exiting.  
Main thread exiting.
```



```
System.out.println();

// Set the name and priority.
System.out.println("Setting name and priority.\n");
thrd.setName("Thread #1");
thrd.setPriority(Thread.NORM_PRIORITY+3);

System.out.println("Main thread is now called: " +
                   thrd.getName());

System.out.println("Priority is now: " +
                   thrd.getPriority());
    }
}
```