Multithreading

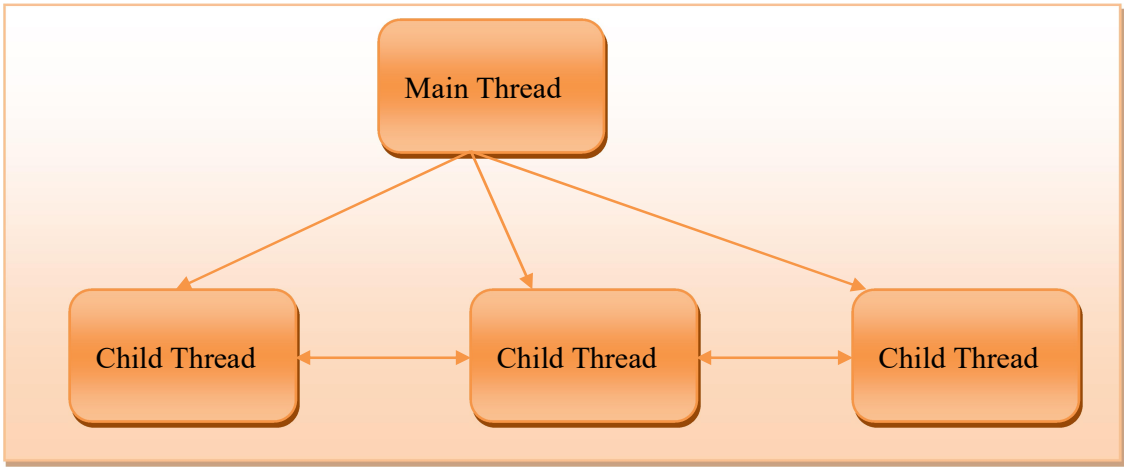
A multithreaded program contains two or more parts that can run on currently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Multithreading is a specialized form of multitasking.

There are two distinct types of multitasking: process-based and thread-based. **A *process* is, in essence, a program that is executing.**

1. 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.

2. 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 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.

Java Thread Model



Messaging

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. The ones that will be used in this chapter are shown here:

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.
- It must be the last thread to finish execution. When the main thread stops, your program terminates.

The main thread is created automatically when your program is started, it can be controlled through a **Thread** object. 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. Let's begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
t.setName("My Thread");// change the name of the 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 wouldhappen 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
```

Creating a Thread

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:

1. You can implement the **Runnable** interface.
2. You can extend the **Thread** class, itself.

Implementing Runnable:

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, until their loops finish. The output produced by this program is as follows:

```
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.
```

NOTE: If the main thread finishes before a child thread has completed, then the Java run-time system may "hang."

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.

Creating Multiple Threads:

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.");
}
}
```

The output from this program is shown here:

```
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.
```

Where, 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.

Using isAlive( ) and join( )

Que:How can one thread know when another thread has ended?
Thread provides a means by which you can answer this question. Two ways exist to determine whether a thread has finished.

1. 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.
2. 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:

```
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.
```

where, after the calls to **join( )** return, the threads have stopped executing.

Thread Priorities:

"Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. "

- In theory, 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.
- 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.

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.

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 **final** variables within **Thread**. You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

final int getPriority( )

The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a non-preemptive platform. One thread is set two levels above the normal priority, as defined by **Thread.NORM_PRIORITY**, and the other is set to two levels below it.

The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

```
// Demonstrate thread priorities.
class clicker implements Runnable {
int click = 0;
Thread t;
private volatile boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
```

```
public void run() {
while (running) {
click++;
}
}
public void stop() {
running = false;
}
public void start() {
t.start();
}}
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
hi.t.join();
lo.t.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
}}
```

The output of this program, shown as follows when run under Windows 98, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got approximately 90 percent of the CPU time.

Low-priority thread: 4408112
High-priority thread: 589626904