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.

Multithreaded Fundamentals

Process vs Thread

- A process is, in essence, 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 while you are using a text editor or visiting a web site.
- 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 while it is printing, if these two actions are being performed by two separate threads.
- Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.
- While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java’s control. However, multithreaded multitasking is.
- Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system.

Thread 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.

The Thread Class and Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it.

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.

```java
//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");
		}
	}
}
```

Creating a Thread

Java defines two ways in which this can be accomplished:

- Implement the Runnable interface.
- Extend the Thread class, itself.

Implementing Runnable

```
//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.");
	}
}


public class Test {
	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.");
	}
}
```

Extending 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.");
	}
}


public class Test {
	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.");
	}
}
```

Creating Multiple 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.");
	}
}


public class Test {
	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.");
	}
}
```

Determining When a Thread Ends

Two ways exist to determine whether a thread has finished: isAlive() and join() methods.

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

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

```
public class Test {
	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.");
	}
}
```

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.

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.

To set a thread's priority, use the setPriority( ) method, which is a member of Thread.

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.

final int getPriority( )

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*.

- 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.

Using synchronized Statement

```
//This program uses a synchronized block.
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();
	}

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


public class Test {
	public static void main(String args[]) {
		Callme target = new Callme();
		Caller ob1 = new Caller(target, "Hello");
		Caller ob2 = new Caller(target, "Synchronized");
		Caller ob3 = new Caller(target, "World");
		// wait for threads to end
		try {
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
		} catch (InterruptedException e) {
			System.out.println("Interrupted");
		}
	}
}
```

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

To avoid polling, Java includes an elegant interprocess 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.

- 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.

```
//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();
		}
	}
}


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

Suspending, Resuming, and Stopping Threads

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and 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. However, all these methods are **deprecated** by Java 2, since these methods can sometimes cause serious system failures.

Because you can't now use the suspend( ), resume( ), or stop( ) methods to control a thread, you might be thinking that no way exists 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 whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this 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. If it is set to "stop," the thread must terminate.

The following example illustrates how the wait( ) and notify( ) methods that are inherited from Object can be used to control the execution of a thread. Let us consider its operation.

- The NewThread class contains a boolean instance variable named suspendFlag, which is used to control the execution of the thread.
- It is initialized to false by the constructor.
- The run( ) method contains a synchronized statement block that checks suspendFlag.
- If that variable is true, the wait( ) method is invoked to suspend the execution of the thread.
- The mysuspend( ) method sets suspendFlag to true. The myresume( ) method sets suspendFlag to false and invokes notify( ) to wake up the thread.
- Finally, the main( ) method has been modified to invoke the mysuspend( ) and myresume( ) methods.

```
//Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
	String name; // name of thread
	Thread t;
	boolean suspendFlag;

	NewThread(String threadname) {
		name = threadname;
		t = new Thread(this, name);
		System.out.println("New thread: " + t);
		suspendFlag = false;
		t.start(); // Start the thread
	}

	//This is the entry point for thread.
	public void run() {
		try {
			for (int i = 15; i > 0; i--) {
```

```
                        System.out.println(name + ": " + i);
                        Thread.sleep(200);
                        synchronized (this) {
                              while (suspendFlag) {
                                    wait();
                              }
                        }
                  }
            } catch (InterruptedException e) {
                  System.out.println(name + " interrupted.");
            }
            System.out.println(name + " exiting.");
      }

      synchronized void mysuspend() {
            suspendFlag = true;
      }

      synchronized void myresume() {
            suspendFlag = false;
            notify();
      }
}


public class Test {
      public static void main(String args[]) {
            NewThread ob1 = new NewThread("One");
            NewThread ob2 = new NewThread("Two");
            try {
                  Thread.sleep(1000);
                  ob1.mysuspend();
                  System.out.println("Suspending thread One");
                  Thread.sleep(1000);
                  ob1.myresume();
                  System.out.println("Resuming thread One");
                  ob2.mysuspend();
                  System.out.println("Suspending thread Two");
                  Thread.sleep(1000);
                  ob2.myresume();
                  System.out.println("Resuming thread Two");
            } catch (InterruptedException e) {
                  System.out.println("Main thread Interrupted");
            }
            // wait for threads to finish
            try {
                  System.out.println("Waiting for threads to finish.");
                  ob1.t.join();
                  ob2.t.join();
            } catch (InterruptedException e) {
                  System.out.println("Main thread Interrupted");
            }
            System.out.println("Main thread exiting.");
      }
}
```