MODULE:IV

What are threads?

- 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.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. Multitasking threads require less overhead than multitasking processes.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- The **Thread** class defines several methods that help manage threads (shown below)

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.

Output:

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

These 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.

The general form of sleep() is:

static void sleep(long milliseconds) throws InterruptedException

The number of milliseconds to suspend is specified in milliseconds. This method may throw an **InterruptedException**.

Creating a Thread

There are two different ways to create threads.

- You can implement the **Runnable** interface.
- 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. 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( )

run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

Thread defines several constructors.

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( )

Example:

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

Output:

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.

a new **Thread** object is created by the following statement:

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

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.

Extending Thread Class

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.

Example:

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

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

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.
```

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.

Using isAlive( ) and join( )

To make main to finish last 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.
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.

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

Output:

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.

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. Inpractice, the amount of CPU time that a thread gets often depends on several factors besides its priority. 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( )

The following example demonstrates two threads at different priorities, 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 madeit through the loop is then displayed.

```java
// Demonstrate thread priorities.
class clicker implements Runnable {
  long 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 higher-priority thread got the majority of the CPU time.
Low-priority thread: 4408112

High-priority thread: 589626904

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 (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. 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.

Using Synchronized Methods

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.

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();
      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");
      }
   }
}
```

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.

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) {
...
```

After **synchronized** has been added to **call( )**,
the output of the program is as follows:
[Hello]
[Synchronized]
[World]

The synchronized Statement

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

Here, object is a reference to the object being synchronized.
Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```java
// 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);
    }
  }
}

class Synch1 {
  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");
    }
  }
}
```

Interthread Communication

Java supports interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.

- **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( )**.
- **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 notifyAll( )

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.**

```java
// 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.

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:

```java
// 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 the **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 the **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 the **Consumer** that it should now remove it.

Here is some output from this program:

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

Got: 3

Put: 4

Got: 4

Put: 5

Got: 5

Event Handling

The Delegation Event Model

- *Delegation event model* defines standard and consistent mechanisms to generate and process events.
- A *source* generates an event and sends it to one or more *listeners.* In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

Events:

- An *event* is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sources:

- A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method. General form:

    public void add*Type*Listener(*Type*Listener *el*)

- Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**.
- A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

    public void remove*Type*Listener(*Type*Listener *el*)

- Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

Event Listeners
- A *listener* is an object that is notified when an event occurs.
- It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

Event Classes

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-1 Main Event Classes in **java.awt.event**

ActionEvent Class

- An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a
- menu item is selected.
- **ActionEvent** has these three constructors:
  ActionEvent(Object *src*, int *type*, String *cmd*)
  ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
  ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)
- Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type,* and its command string is *cmd.* The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred.
- You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:
  String getActionCommand( )

- The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:
  int getModifiers( )
- The method **getWhen( )** returns the time at which the event took place. This is called the event's *timestamp.* The **getWhen( )** method is shown here:
  long getWhen( )

AdjustmentEvent Class

- An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events.

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

- **AdjustmentEvent** constructor:
  AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *data*)
  Here, *src* is a reference to the object that generated this event. The *id* specifies the event. The type of the adjustment is specified by *type,* and its associated data is *data.*

ComponentEvent Class

- A **ComponentEvent** is generated when the size, position, or visibility of a component is changed.
- There are four types of component events.

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

- **ComponentEvent** has this constructor:
  ComponentEvent(Component *src*, int *type*)
  Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*
- **ComponentEvent** is the superclass either directly or indirectly of ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent.
- The **getComponent( )** method returns the component that generated the event. It is shown here:
  Component getComponent( )

ContainerEvent Class

- A **ContainerEvent** is generated when a component is added to or removed from a container.
- There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**.
- **ContainerEvent** is a subclass of **ComponentEvent** and has this constructor:
  ContainerEvent(Component *src*, int *type*, Component *comp*)
  Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*.
- You can obtain a reference to the container that generated this event by using the **getContainer( )** method, shown here:
  Container getContainer( )
- The **getChild( )** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:
  Component getChild( )

FocusEvent Class

- A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.
- **FocusEvent** is a subclass of **ComponentEvent** and has these constructors:
  FocusEvent(Component *src*, int *type*)
  FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
  FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)
  Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**.
- You can determine the other component by calling **getOppositeComponent( )**, shown here:
  Component getOppositeComponent( )
  The opposite component is returned.
- The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:
  boolean isTemporary( )
  The method returns **true** if the change is temporary. Otherwise, it returns **false**.

InputEvent Class

- It is the superclass for component input events.
- Its subclasses are **KeyEvent** and **MouseEvent**.
- **InputEvent** defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event.

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

- To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:
  boolean isAltDown( )
  boolean isAltGraphDown( )
  boolean isControlDown( )
  boolean isMetaDown( )
  boolean isShiftDown( )
- You can obtain a value that contains all of the original modifier flags by calling the **getModifiers( )** method. It is shown here:
  int getModifiers( )

ItemEvent Class

- An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

- **ItemEvent** has this constructor:
  ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)
  Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.
- The **getItem( )** method can be used to obtain a reference to the item that generated an event. Its signature is shown here:
  Object getItem( )
- The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:
  temSelectable getItemSelectable( )

KeyEvent Class

- A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and**KEY_TYPED**.

- There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

- The **VK** constants specify *virtual key codes*
- **KeyEvent** is a subclass of **InputEvent**. Here is one of its constructors:
  KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)
  Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*
- **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:
  char getKeyChar( )
  int getKeyCode( )
- If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

MouseEvent Class

- There are eight types of mouse events.

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

- **MouseEvent** is a subclass of **InputEvent**. Here is one of its constructors:
  MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*)
  Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *odifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.
- **getX( )** and **getY( ):** These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:
  int getX( )
  int getY( )

- **getPoint( )** method to obtain the coordinates of the mouse.
  Point getPoint( )
- The **translatePoint( )** method changes the location of the event. Its form is shown here:
  void translatePoint(int *x*, int *y*)
  Here, the arguments *x* and *y* are added to the coordinates of the event.
- The **getClickCount( )** method obtains the number of mouse clicks for this event.
  Its signature is shown here:
  int getClickCount( )
- The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:
  boolean isPopupTrigger( )
- **getButton( )** method, shown here:
  int getButton( )
  It returns a value that represents the button that caused the event. The return value will be one of these constants defined by **MouseEvent**:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

MouseWheelEvent Class

- The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**.
- **MouseWheelEvent** defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

- Here is one of the constructors defined by **MouseWheelEvent**:
  MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*, int *scrollHow*, int *amount*, int *count*)
  Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred.
- To obtain the number of rotational units, call **getWheelRotation( )**, shown here:
  int getWheelRotation( )

TextEvent Class

- These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.
- The one constructor for this class is shown here:
  TextEvent(Object *src*, int *type*)
  Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

WindowEvent Class

- There are ten types of window events.

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

- **WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors.
  WindowEvent(Window *src*, int *type*)
  WindowEvent(Window *src*, int *type*, Window *other*)
  WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
  WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)
  other specifies the opposite window when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *toState* specifies the new state that the window will have when a window state change occurs.

- **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:
  Window getWindow( )

Sources of Events

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-3   Commonly Used Event Listener Interfaces

ActionListener Interface
This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:
        void actionPerformed(ActionEvent *ae*)

AdjustmentListener Interface
This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:
        void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface
This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:
        void componentResized(ComponentEvent *ce*)
        void componentMoved(ComponentEvent *ce*)
        void componentShown(ComponentEvent *ce*)
        void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface
This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

      void componentAdded(ContainerEvent *ce*)

      void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface
This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

      void focusGained(FocusEvent *fe*)
      void focusLost(FocusEvent *fe*)

The ItemListener Interface
This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

      void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface
This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.
The general forms of these methods are shown here:

      void keyPressed(KeyEvent *ke*)
      void keyReleased(KeyEvent *ke*)
      void keyTyped(KeyEvent *ke*)

The MouseListener Interface
This interface defines five methods.
The general forms of these methods are shown here:

      void mouseClicked(MouseEvent *me*)
      void mouseEntered(MouseEvent *me*)
      void mouseExited(MouseEvent *me*)
      void mousePressed(MouseEvent *me*)
      void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface
This interface defines two methods.

      void mouseDragged(MouseEvent *me*)
      void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface
This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

      void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface
This interface defines the **textChanged( )** method that is invoked when a change occurs
in a text area or text field. Its general form is shown here:
        void textChanged(TextEvent *te)*

The WindowFocusListener Interface
This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These
are called when a window gains or loses input focus. Their general forms are shown here:
        void windowGainedFocus(WindowEvent *we)*
        void windowLostFocus(WindowEvent *we)*

The WindowListener Interface
This interface defines seven methods.
        void windowActivated(WindowEvent *we)*
        void windowClosed(WindowEvent *we)*
        void windowClosing(WindowEvent *we)*
        void windowDeactivated(WindowEvent *we)*
        void windowDeiconified(WindowEvent *we)*
        void windowIconified(WindowEvent *we)*
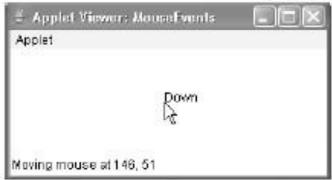   void windowOpened(WindowEvent *we)*

Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the
MouseMotionListener interfaces.

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener
{
        String msg = "";
        int mouseX = 0, mouseY = 0; // coordinates of mouse
        public void init()
        {
                addMouseListener(this);
                addMouseMotionListener(this);
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
         {
```

```java
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked.";
        repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me)
{
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse entered.";
        repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me)
{
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse exited.";
        repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me)
{
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down";
        repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me)
{
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Up";
        repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
        // save coordinates
        mouseX = me.getX();
```

```
                mouseY = me.getY();
                msg = "*";
                showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
                repaint();
        }
// Handle mouse moved.
public void mouseMoved(MouseEvent me)
{
        // show status
        showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}
// Display msg in applet window at current X,Y location.
public void paint(Graphics g)
{
        g.drawString(msg, mouseX, mouseY);
}
}
```


- It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
- It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
- The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces.
- Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here:

void addMouseListener(MouseListener *ml*)
void addMouseMotionListener(MouseMotionListener *mml*)

Handling Keyboard Events

- When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler.
- When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed.
- If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked.

```java
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet implements KeyListener
{
        String msg = "";
        int X = 10, Y = 20; // output coordinates
        public void init()
        {
                addKeyListener(this);
        }
        public void keyPressed(KeyEvent ke)
        {
                showStatus("Key Down");
        }
        public void keyReleased(KeyEvent ke)
        {
                showStatus("Key Up");
        }
        public void keyTyped(KeyEvent ke)
        {
                msg += ke.getKeyChar();
        }

        // Display keystrokes.
        public void paint(Graphics g)
        {
                g.drawString(msg, X, Y);
        }
}
```

Adapter Classes

An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

```
// Demonstrate an
adapter. import
java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet
{
        public void init()
        {
                addMouseListener(new MyMouseAdapter(this));
                addMouseMotionListener(new MyMouseMotionAdapter(this));
        }
}

class MyMouseAdapter extends MouseAdapter
{
        AdapterDemo adapterDemo;
        public MyMouseAdapter(AdapterDemo adapterDemo)
```

```
        {
                this.adapterDemo = adapterDemo;
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {
                adapterDemo.showStatus("Mouse clicked");
        }
}

class MyMouseMotionAdapter extends MouseMotionAdapter
{
        AdapterDemo adapterDemo;
        public MyMouseMotionAdapter(AdapterDemo adapterDemo)
        {
                this.adapterDemo = adapterDemo;
        }
        // Handle mouse dragged.
        public void mouseDragged(MouseEvent me)
        {
                adapterDemo.showStatus("Mouse dragged");
        }
}
```

- It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored.
- The program has three classes.
- **AdapterDemo** extends **Applet**. Its **init( )** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events.
- **MyMouseAdapter** extends **MouseAdapter** and overrides the **mouseClicked( )** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class.
- **MyMouseMotionAdapter** extends **MouseMotionAdapter** and overrides the **mouseDragged( )** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

Inner Classes

An *inner class* is a class defined within another class, or even within an expression.

```
// Inner class demo.
import java.applet.*;
import
java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/
public class InnerClassDemo extends Applet
{
        public void init()
        {
                addMouseListener(new MyMouseAdapter());
        }
        class MyMouseAdapter extends MouseAdapter
        {
                public void mousePressed(MouseEvent me)
                {
                        showStatus("Mouse Pressed");
                }
        }
}
```

- Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**.
- Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed( )** method can call the **showStatus( )** method directly.

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name.

```
// Anonymous inner class
demo. import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet
{
        public void init()
        {
                addMouseListener(new MouseAdapter()
                {
                        public void mousePressed(MouseEvent me)
                        {
                        showStatus("Mouse Pressed");
                        }
                });
        }
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**.

The **init( )** method calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class.