MODULE-5
Chapter - 1
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**.
- However, there are two distinct types of multitasking: **process-based and thread-based**.

- It is important to understand the difference between the two.
- **For many readers, process-based multitasking is the more familiar form.**
- **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 at the same time that 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.**
- Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system.
- One important way multithreading achieves this is by keeping idle time to a minimum.
- Multithreading helps you reduce this idle time because another thread can run when one is waiting.

The Java Thread Model

- **The Java Thread Model The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.**
- **In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.**

- **Single-threaded systems use an approach called an event loop with polling.**
- **In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.**
- **Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler.**
- **Until this event handler returns, nothing else can happen in the program.**
- **This wastes CPU time.**
- **It can also result in one part of a program dominating the system and preventing any other events from being processed.**
- **In general, in a single-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.**
- **The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program.**
- **Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause.**
- When a thread blocks in a Java program, **only the single thread that is blocked pauses.**
- **All other threads continue to run.**
- As most readers know, over the past few years, multi-core systems have become common place. Of course, single-core systems are still in widespread use.

- It is important to understand that Java's multithreading features work in both types of systems.
- **In a single core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time.**
- **Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized**.
- However, in multi-core systems, it is possible **for two or more threads to actually execute simultaneously.**
- **In many cases, this can further improve program efficiency and increase the speed of certain operations**.
- It provides a powerful means of creating multithreaded applications that automatically scale to make best use of multi-core environments.
- **The Fork/Join Framework is part of Java's support for parallel programming, which is the name commonly given to the techniques that optimize some types of algorithms for parallel execution in systems that have more than one CPU**.
- Threads exist in several states. **Here is a general description. A thread can be running. It can be ready to run as soon as it gets CPU time.**
- A running thread can be suspended, which temporarily halts its activity.
- A suspended thread can then be resumed, allowing it to pick up where it left off.
- A thread can be blocked when waiting for a resource.
- At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

- Java assigns to each thread a priority **that determines how that thread should be treated with respect to the others.**
- **Thread priorities are integers that specify the relative priority of one thread to another**.
- a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
- **Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.**
- The rules that determine when a context switch takes place are simple:
  - **A thread can voluntarily relinquish control:** This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
  - **A thread can be preempted by a higher-priority thread:** In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. **Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.**

In cases where two threads with **the same priority are competing for CPU cycles, the situation is a bit complicated.**

For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

The Thread Class and the 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. Several of those 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.

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and **Runnable** to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

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:**
    1. It is the thread from which other "child" threads will be spawned.
    2. Often, it must be the last thread to finish execution because it performs various shutdown actions.
    3. Although the main thread **is created automatically when your program is started, it can be controlled through a Thread object**.
    4. To do so, you must obtain a reference to it by **calling the method currentThread( ), which is a public static member of Thread.**
    5. **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);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

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

thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

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

of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

static void sleep(long *milliseconds*) throws InterruptedException

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

The **sleep( )** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

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

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using **setName( )**. You can obtain the name of a thread by calling **getName( )** (but note that this is not shown in the program). These methods are members of the **Thread** class and are declared like this:

final void setName(String *threadName*)
final String getName( )

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

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:

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

The following two sections look at each method, in turn.

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 in single-core systems, until their loops finish. The output produced by this program is as follows. (Your output may vary based upon the specific execution environment.)

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

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**.

Creating Multiple Threads

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

```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

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

  // This is the entry point for thread.
  public void run() {
   try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
   } catch (InterruptedException e) {
     System.out.println(name + "Interrupted");
   }
   System.out.println(name + " exiting.");
  }
}
```

```
class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");

    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

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

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using isAlive( ) and join( )

- As mentioned, often you **will want the main thread to finish last.**
- In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread
- However, this is hardly satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, Thread provides a means by which you can answer this question.
- **Two ways exist to determine whether a thread has finished.**
- **First, you can call isAlive( ) on the thread.**
- **This method is defined by Thread, and its general form is shown here:**

  final boolean isAlive( )
- **The isAlive( ) method returns true if the thread upon which it is called is still running.**
- **It returns false otherwise.**
- **While isAlive( ) is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called join( ), shown here:**

  final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates.
- Its name **comes from the concept of the calling thread waiting until the specified thread joins it.**
- **Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.**

Here is an improved version of the preceding example that uses **join( )** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive( )** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

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

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}

class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");
```

```
        System.out.println("Thread One is alive: "
                           + ob1.t.isAlive());
        System.out.println("Thread Two is alive: "
                           + ob2.t.isAlive());
        System.out.println("Thread Three is alive: "
                           + ob3.t.isAlive());
        // wait for threads to finish
        try {
          System.out.println("Waiting for threads to finish.");
          ob1.t.join();
          ob2.t.join();
          ob3.t.join();
        } catch (InterruptedException e) {
          System.out.println("Main thread Interrupted");
        }

        System.out.println("Thread One is alive: "
                           + ob1.t.isAlive());
        System.out.println("Thread Two is alive: "
                           + ob2.t.isAlive());
        System.out.println("Thread Three is alive: "
                           + ob3.t.isAlive());

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

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, over a given period of time, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)
- A higher-priority thread can also **preempt a lower-priority one.**
- **For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.**
- In theory, threads of equal priority should get equal access to the CPU. But you need to be careful.
- Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others.

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**. This is its general form:

final void setPriority(int *level*)

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

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

final int getPriority( )

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

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.
- **You can synchronize your code in either of two ways. Both involve the use of the synchronized keyword, and both are examined here.**

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- **To enter an object's monitor, just call a method that has been modified with the synchronized keyword**.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- let's begin with a simple example that does not use it—but should.

```
// 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. This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

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

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

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

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

The synchronized Statement

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

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

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

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

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

```
// 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

- As you will see, this is especially easy in Java.
- As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units.
- Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly.
- Once the condition is true, appropriate action is taken.
- This wastes CPU time.

- For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it.
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce.
- Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.
- Clearly, this situation is undesirable.
- To avoid polling, Java includes an elegant inter process 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.
- Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

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

These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )
```

Additional forms of **wait( )** exist that allow you to specify a period of time to wait.

- **Before working through an example that illustrates interthread communication, an important point needs to be made**.

- Although **wait( ) normally waits until notify( ) or notifyAll( ) is called, there is a possibility that in very rare cases the waiting thread could be awakened(stop of sleep) due to a spurious wakeup.(without satisfaction)**
- In this case, a waiting thread resumes without **notify( ) or notifyAll( ) having been called.**
- (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, Oracle recommends that calls to wait( ) should take place within a loop that checks the condition on which the thread is waiting.

```
// An incorrect implementation of a producer and consumer.
class Q {
  int n;

  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

   while(true) {
     q.put(i++);
   }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}

class PC {
  public static void main(String args[]) {
```

```
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

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

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


Deadlock

- **A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.**
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
- If the thread in X tries to call any synchronized method on Y, it will block as expected.

- However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.
- **Deadlock is a difficult error to debug for two reasons**:
  - In general, it occurs only rarely, when the two threads time-slice in just the right way.
  - It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

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

```
// An example of deadlock.
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();

    System.out.println(name + " entered A.foo");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrupted");
```

```
      System.out.println(name + " trying to call B.last()");
      b.last();
    }

    synchronized void last() {
      System.out.println("Inside A.last");
    }
  }

  class B {
    synchronized void bar(A a) {
      String name = Thread.currentThread().getName();
      System.out.println(name + " entered B.bar");

      try {
        Thread.sleep(1000);
      } catch(Exception e) {
        System.out.println("B Interrupted");
      }
          }

        System.out.println(name + " trying to call A.last()");
        a.last();
      }

      synchronized void last() {
        System.out.println("Inside A.last");
      }
    }

    class Deadlock implements Runnable {
      A a = new A();
      B b = new B();

      Deadlock() {
        Thread.currentThread().setName("MainThread");
        Thread t = new Thread(this, "RacingThread");
        t.start();

        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
      }
```

```
  public void run() {
    b.bar(a); // get lock on b in other thread.
    System.out.println("Back in other thread");
  }

  public static void main(String args[]) {
    new Deadlock();
  }
}
```

When you run this program, you will see the output shown here:

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

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC. You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

Suspending, Resuming, and Stopping Threads

- Sometimes, **suspending execution of a thread is useful.**
- **For example, a separate thread can be used to display the time of day.**
- **If the user doesn't want a clock, then its thread can be suspended.**
- Whatever the case, suspending a thread is a simple matter.

- Once suspended, restarting the thread is also a simple matter.
- 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.**
- **The suspend( ) method of the Thread class was deprecated by Java 2 several years ago.**
- **This was done because suspend( ) can sometimes cause serious system failures.**
- **Assume that a thread has obtained locks on critical data structures.**
- If that thread is suspended at that point, those locks are not relinquished.(retreat)
- Other threads that may be waiting for those resources can be deadlocked.
- **The resume( ) method is also deprecated.**
- It does not cause problems, but cannot be used without the suspend( ) method as its counterpart.
- **The stop( ) method** of the Thread class, too, was deprecated by Java 2.
- This was done because this method can sometimes cause serious system failures.
- Assume that a thread is writing to a critically important data structure and has completed only part of its changes.
- If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that stop( ) causes any lock the calling thread holds to be released.

- **Thus, the corrupted data might be used by another thread that is waiting on the same lock.**
- **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.**

Obtaining A Thread's State

As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the **getState( )** method defined by **Thread**. It is shown here:

```
Thread.State getState( )
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. (An enumeration is a list of named constants. It is discussed in detail in Chapter 12.) Here are the values that can be returned by **getState( )**:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.

Figure 11-1 diagrams how the various thread states relate.

Figure 11-1 diagrams how the various thread states relate.

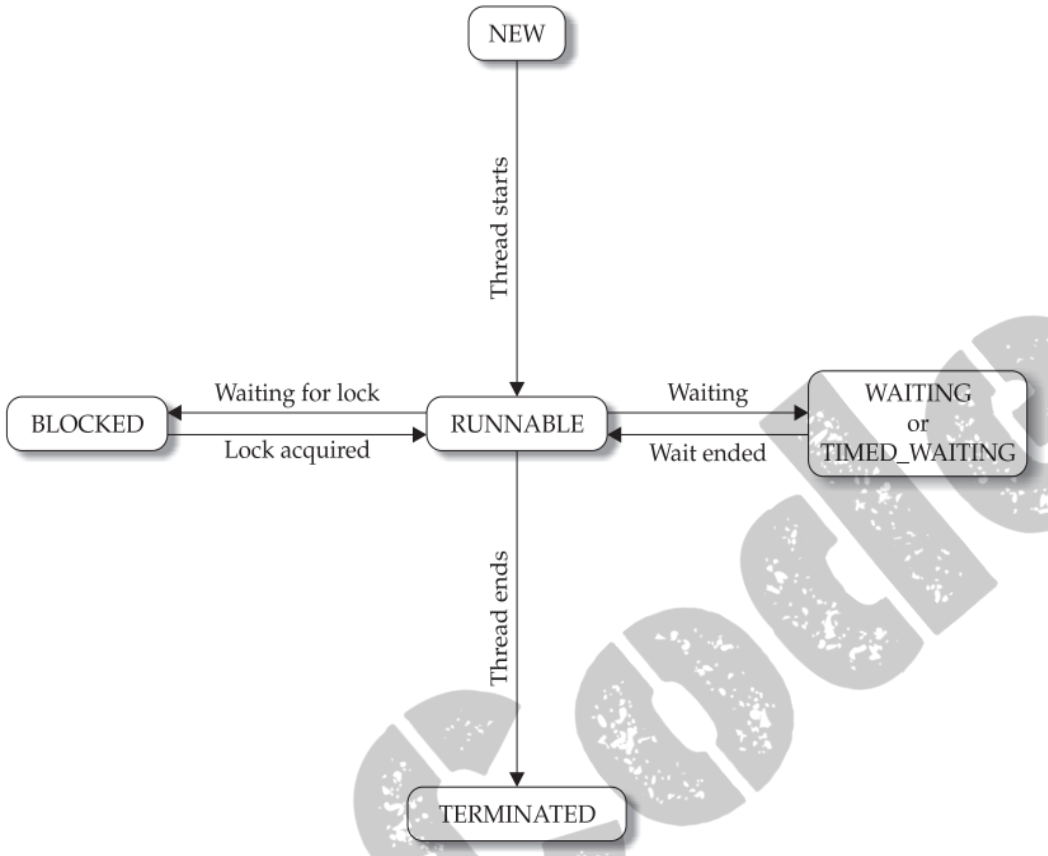

Figure 11-1 Thread states

Given a **Thread** instance, you can use **getState( )** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState( )** is called:

```
Thread.State ts = thrd.getState();

if(ts == Thread.State.RUNNABLE) // ...
```

It is important to understand that a thread's state may change after the call to **getState( )**. Thus, depending on the circumstances, the state obtained by calling **getState( )** may not reflect the actual state of the thread only a moment later. For this (and other) reasons, **getState( )** is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

MODULE-5
CHAPTER-2
Enumerations, Type Wrappers and Autoboxing

Enumerations

- **Versions of Java prior to JDK 5 lacked one feature that many programmers felt was needed: enumerations.**
- **In its simplest form, an enumeration is a list of named constants.**
- Although Java offered other features that provide somewhat similar functionality, such as final variables, many programmers still missed the conceptual purity of enumerations— especially because enumerations are supported by many other commonly used languages.
- Beginning with JDK 5, enumerations were added to the Java language, and they are now an integral and widely used part of Java.
- **In their simplest form, Java enumerations appear similar to enumerations in other languages. However, this similarity may be only skin deep because, in Java, an enumeration defines a class type.**
- By making enumerations into classes, the capabilities of the enumeration are greatly expanded.
- For example, in Java, an enumeration can have constructors, methods, and instance variables.

Enumeration Fundamentals

An enumeration is created using the **enum** keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

The identifiers **Jonathan**, **GoldenDel**, and so on, are called *enumeration constants*. Each is implicitly declared as a public, static final member of **Apple**. Furthermore, their type is the type of the enumeration in which they are declared, which is **Apple** in this case. Thus, in the language of Java, these constants are called *self-typed*, in which "self" refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new**. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares **ap** as a variable of enumeration type **Apple**:

```
Apple ap;
```

Because **ap** is of type **Apple**, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns **ap** the value **RedDel**:

```
ap = Apple.RedDel;
```

Notice that the symbol **RedDel** is preceded by **Apple**.

Two enumeration constants can be compared for equality by using the = = relational operator. For example, this statement compares the value in **ap** with the **GoldenDel** constant:

```
if(ap == Apple.GoldenDel) // ...
```

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(ap) {
  case Jonathan:
    // ...
  case Winesap:
    // ...
```

The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: **values**( ) and **valueOf**( ). Their general forms are shown here:

public static *enum-type* [ ] values( )
public static *enum-type* valueOf(String *str*)

- **The values( ) method returns an array that contains a list of the enumeration constants.**
- **The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.**
- **In both cases, enum-type is the type of the enumeration.**

The following program demonstrates the **values( )** and **valueOf( )** methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Apple ap;

    System.out.println("Here are all Apple constants:");

    // use values()
    Apple allapples[] = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);

    System.out.println();

    // use valueOf()
    ap = Apple.valueOf("Winesap");
    System.out.println("ap contains " + ap);

  }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Type Wrappers

- **As you know, Java uses primitive types (also called simple types), such as int or double, to hold the basic data types supported by the language.**
- the primitive types are not part of the object hierarchy, and they do not inherit Object.
- Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.
- **For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.**
- **To handle these (and other) situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.**
- **The type wrapper classes are described in detail in Part II, but they are introduced here because they relate directly to Java's autoboxing feature.**
- **The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean.**
- **These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.**

Character

Character is a wrapper around a **char**. The constructor for **Character** is

Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here

char charValue( )

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:

Boolean(boolean *boolValue*)
Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here:

boolean booleanValue( )

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

- By far, the most commonly used type wrappers are those that represent numeric values.
- **These are Byte, Short, Integer, Long, Float, and Double.**
- **All of the numeric type wrappers inherit the abstract class Number.**
- **Number declares methods that return the value of an object in each of the different number formats.**
- **These methods are shown here:**

byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

Integer(int *num*)
Integer(String *str*)

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.
class Wrap {
  public static void main(String args[]) {

    Integer iOb = new Integer(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); // displays 100 100
  }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

Autoboxing

- **Beginning with JDK 5, Java added two important features: autoboxing and auto-unboxing.**
- **Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.**
- There is no need to explicitly construct an object.
- **Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.**
- There is no need to call a method such as **intValue( ) or doubleValue( ).**
- The addition of autoboxing and auto-unboxing greatly streamlines the **coding of several algorithms, removing the tedium of manually boxing and unboxing values.**
- It also **helps prevent errors.**
- Moreover, it is very important to generics, which **operate only on objects.**
- Finally, autoboxing makes working with the **Collections Framework.**
- With autoboxing, it is no longer necessary to manually construct an object in order to wrap a primitive type.
- You need only assign that value to a type-wrapper reference.
- Java automatically constructs the object for you.
- **For example, here is the modern way to construct an Integer object that has the value 100:**

```
Integer iOb = 100; // autobox an int
```

Notice that the object is not explicitly created through the use of **new**. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
  public static void main(String args[]) {

    Integer iOb = 100; // autobox an int

    int i = iOb; // auto-unbox

    System.out.println(i + " " + iOb);  // displays 100 100
  }
}
```


Autoboxing and Methods

- In addition to the simple case of assignments**, autoboxing automatically occurs whenever a primitive type must be converted into an object;**
- auto-unboxing takes place whenever an object must be converted into a primitive type.
- Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
  // Take an Integer parameter and return
  // an int value;
  static int m(Integer v) {
    return v ; // auto-unbox to int
  }

  public static void main(String args[]) {
    // Pass an int to m() and assign the return value
    // to an Integer.  Here, the argument 100 is autoboxed
    // into an Integer.  The return value is also autoboxed
    // into an Integer.
    Integer iOb = m(100);

    System.out.println(iOb);
  }
}
```

This program displays the following result:

```
100
```

Autoboxing/Unboxing Occurs in Expressions

- **In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required.**
- **This applies to expressions.**
- **Within an expression, a numeric object is automatically unboxed.**
- **The outcome of the expression is reboxed, if necessary.**
- **For example, consider the following program:**

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);

    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);

    // The same expression is evaluated, but the
    // result is not reboxed.
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);

  }
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
  public static void main(String args[]) {

    // Autobox/unbox a boolean.
    Boolean b = true;

    // Below, b is auto-unboxed when used in
    // a conditional expression, such as an if.
    if(b) System.out.println("b is true");

    // Autobox/unbox a char.
    Character ch = 'x'; // box a char
    char ch2 = ch; // unbox a char

    System.out.println("ch2 is " + ch2);
  }
}
```

The output is shown here:

```
b is true
ch2 is x
```

The most important thing to notice about this program is the auto-unboxing of **b** inside the **if** conditional expression. As you should recall, the conditional expression that controls an **if** must evaluate to type **boolean**. Because of auto-unboxing, the **boolean** value contained within **b** is automatically unboxed when the conditional expression is evaluated. Thus, with the advent of autoboxing/unboxing, a **Boolean** object can be used to control an **if** statement.

Because of auto-unboxing, a **Boolean** object can now also be used to control any of Java's loop statements. When a **Boolean** is used as the conditional expression of a **while**, **for**, or **do/while**, it is automatically unboxed into its **boolean** equivalent. For example, this is now perfectly valid code:

```
Boolean b;
// ...
while(b) { // ...
```