

---

# OBJECT ORIENTED PROGRAMMING WITH JAVA

---

BCS306A

---

*Module V*  
*Notes*

---

JANUARY 25, 2024  
CANARA ENGINEERING COLLEGE  
Sudhindra Nagar, Benjanapadavu

**Module V –  
Multithreaded Programming, Enumerations, Type Wrappers and  
Autoboxing**

**Table of Contents**

5.1 Introduction.....	1
5.2 The Java Thread Model.....	1
5.2.1 Thread Priorities.....	2
5.2.2 Synchronization: .....	3
5.2.3 Messaging: .....	3
5.2.4 The Thread Class and the Runnable Interface .....	3
5.3 The Main Thread.....	4
5.4 Creating a Thread.....	5
5.4.1 Implementing Runnable.....	6
5.4.2 Extending Thread.....	8
5.4.3 Choosing an Approach.....	9
5.5 Creating Multiple Threads .....	9
5.6 Using isAlive( ) and join( ) .....	11
5.7 Thread Priorities.....	13
5.8 Synchronization .....	14
5.8.1 Using Synchronized Methods .....	14
5.8.2 The synchronized Statement .....	16
5.9 Interthread Communication .....	18
5.9.1 Deadlock .....	23
5.10 Suspending, Resuming, and Stopping Threads.....	25
5. 11 Obtaining a Thread's State .....	27
5.12 Enumerations: .....	28
5.12.1 Enumeration Fundamentals .....	29
5.12.2 The values( ) and valueOf( ) Methods .....	31
5.13 Type wrappers .....	31
5.14 Autoboxing.....	33
5.14.1 Autoboxing and Methods.....	33
5.14.2 Autoboxing/Unboxing Occurs in Expressions.....	34
5.14.3 Autoboxing/Unboxing Boolean and Character Values .....	35

## Module V -

### Multithreaded Programming, Enumerations, Type Wrappers and Autoboxing

---

#### Syllabus:

**Multithreaded Programming:** The Java Thread Model, The Main Thread, Creating a Thread, Creating Multiple Threads. Using `isAlive()` and `join()`, Thread Priorities, Synchronization, Interthread Communication, Suspending. Resuming, and Stopping Threads, Obtaining a Thread's State. **Enumerations, Type Wrappers and Autoboxing:** Enumerations Enumeration Fundamentals, The `values()` and `valueOf()` Methods. Type Wrappers (Character, Boolean, The Numeric Type Wrappers). Autoboxing (Autoboxing and Methods, Autoboxing/Unboxing Occurs in Expressions, Autoboxing/Unboxing Boolean and Character Values).

**Chapter 11, 12.**

---

### Multithreaded Programming

#### 5.1 Introduction

Multithreading is a feature in Java that allows for concurrent execution of multiple parts of a program, called threads. It is a specialized form of multitasking, where each thread represents a separate path of execution. In contrast, process-based multitasking enables running multiple programs simultaneously.

Process-based multitasking involves running multiple programs concurrently, with each program being the smallest unit of code dispatched by the scheduler. It requires separate address spaces for each process, and inter-process communication is expensive and limited.

Thread-based multitasking, on the other hand, operates at a finer granularity, where threads are the smallest units of dispatchable code. Threads share the same address space within a process and can perform multiple tasks simultaneously. Interthread communication is inexpensive, and context switching between threads is more efficient compared to processes.

Java utilizes process-based multitasking environments, but it directly controls and supports multithreaded multitasking. Multithreading allows for efficient utilization of system processing power by minimizing idle time. In interactive and networked environments, where idle time is common, multithreading is crucial. It enables tasks to run concurrently, reducing waiting time for slower operations such as network transmission, file system access, or user input.

#### 5.2 The Java Thread Model

The Java runtime system relies on threads for various functionalities, and the class libraries are designed with multithreading in mind. Threads are used in Java to enable asynchronous behavior, reducing inefficiency by preventing wastage of CPU cycles.

In a single-threaded system, an event loop with polling is utilized. A single thread continuously polls an event queue to determine the next action. This approach wastes CPU time, can lead to dominance of one part of the program, and halts program execution when a thread blocks.

Java's multithreading eliminates the need for the main loop/polling mechanism. One thread can pause without affecting other parts of the program. For example, idle time during network data reading or user input can be utilized elsewhere. In a Java program, when a thread blocks, only that specific thread pauses, and other threads continue their execution.

With the prevalence of multicore systems, Java's multithreading features work in both single-core and multicore systems. In a single-core system, threads share the CPU by receiving slices of CPU time, utilizing idle CPU time effectively. In multicore systems, two or more threads can execute simultaneously, further enhancing program efficiency and speeding up certain operations.

It is worth noting that besides the multithreading features, the Fork/Join Framework in Java provides a powerful tool for creating multithreaded applications that scale automatically in multicore environments. This framework is part of Java's support for parallel programming, optimizing algorithms for parallel execution in systems with multiple CPUs.

Threads in Java can exist in various states, including running, ready to run, suspended, blocked when waiting for a resource, and terminated. Resuming a suspended thread allows it to continue from where it left off, while a terminated thread cannot be resumed.

### 5.2.1 Thread Priorities

Java assigns a priority to each thread, determining how it should be treated relative to other threads. Thread priorities are represented by integers, indicating the relative importance of one thread compared to another. However, a thread's priority does not affect its actual speed when it is the only thread running. Instead, thread priority is used to determine when to switch from one running thread to another, which is known as a context switch.

The rules governing context switches are straightforward:

- A thread can voluntarily release control by explicitly yielding, sleeping, or when blocked. At this point, all other threads are examined, and the highest-priority thread 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 preempted by the higher-priority thread, regardless of what the lower-priority thread is currently doing. This mechanism is referred to as preemptive multitasking.

When two threads of equal priority compete for CPU cycles, the behavior depends on the underlying operating system. Some operating systems automatically time-slice threads of equal priority in a round-robin fashion, while others require threads of equal priority to voluntarily yield control to their counterparts. If they do not yield, the other threads will not have an opportunity to run.

**Caution** Portability problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

### 5.2.2 Synchronization:

Multithreading introduces asynchronous behavior to programs, necessitating a means to enforce synchronicity when required. For instance, when two threads need to communicate and share a complex data structure like a linked list, it is essential to prevent conflicts where one thread writes data while another thread is reading it. Java implements a refined version of inter-process synchronization called the monitor to address this. The monitor, which originated from C.A.R. Hoare, can be envisioned as a small box that can accommodate only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits before they can proceed. Monitors can thus safeguard shared resources from simultaneous manipulation by multiple threads.

In Java, there is no explicit "Monitor" class. Instead, every object has an implicit monitor associated with it. This monitor is automatically entered when any synchronized method of the object is called. Consequently, when a thread is inside a synchronized method, no other thread can invoke any other synchronized method on the same object. This built-in support for synchronization in the language allows for clear and concise multithreaded code.

### 5.2.3 Messaging:

After dividing a program into separate threads, defining how they will communicate becomes crucial. In some programming languages, communication between threads relies on the operating system, introducing additional overhead. In contrast, Java provides a clean and efficient approach for two or more threads to communicate through predefined methods available to all objects. Java's messaging system enables a thread to enter a synchronized method on an object and wait there until another thread explicitly notifies it to resume execution.

### 5.2.4 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 we can't directly refer to the ethereal state of a running thread, we will deal with it through its proxy, the Thread instance that spawned it. To create a new thread, we either extend **Thread** or implement the **Runnable** interface. The **Thread** class defines several methods that help manage threads. Several methods 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.



## 5.3 The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our 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.

While the main thread is automatically created upon program startup, it can be managed using a **Thread** object. To accomplish this, you must obtain a reference to the main thread by invoking the method **currentThread()**, which is a public static member of the **Thread** class. Its general form is: **static Thread currentThread( )**

This method returns a reference to the thread in which it is called. Once we have a reference to the main thread, we can control it just like any other thread.

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

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

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

The program obtains a reference to the current thread (the main thread, in this case) using **currentThread()** and stores it in the variable **t**. Information about the thread is displayed, and then the **setName()** method is called to change the thread's internal name. Afterward, the thread information is displayed again.

Following this, a loop counts down from five, pausing for one second between each iteration using the **sleep()** method. The argument passed to **sleep()** specifies the delay period in *milliseconds*. There is a **try/catch** block surrounding this loop to handle a possible **InterruptedException** that can be thrown by the **sleep()** method. If the sleeping thread is

interrupted by another thread, a message is printed in this example. In a real program, a different approach would be used to handle such interruptions. Here is the output:

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

Notice the output produced when **t** is used as an argument to **println()**. The output includes the thread's name, priority, and group name. The default values for the main thread are "main" for the name, 5 for the priority, and "main" for the group name. A thread group is a data structure that controls the state of a collection of threads as a whole. After changing the thread's name, **t** is again output, displaying the updated name.

The program utilizes the **sleep()** method from the **Thread** class to suspend execution for a specified duration. It has two forms, both forms can throw an **InterruptedException**.

- **sleep()** method that accepts the duration in milliseconds

**static void sleep(long milliseconds) throws InterruptedException**

The number of milliseconds to suspend is specified in *milliseconds*.

- **sleep()** method that allows specifying both *milliseconds* and *nanoseconds*.

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

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

The program demonstrates how to set and obtain the thread name using the **setName()** and **getName()** methods respectively. These methods are defined in the **Thread** class and are used to manipulate the name of a thread. These methods are declared as:

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

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

## 5.4 Creating a Thread

In the most general sense, we create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- by implementing the **Runnable** interface.
- by extending the **Thread** class, itself.

### 5.4.1 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. We 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( )**, we 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 our program. The thread will end when **run( )** returns.

After we create a class that implements **Runnable**, we need to instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we 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 new thread's name is specified by *threadName*.

After the new thread is created, it will not start running until we call its **start( )** method, which is declared within **Thread**, i.e: **start( )** initiates 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() {
        t = new Thread(this, "Demo Thread"); // Create a new, second thread
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }

    public void run() { // This is the entry point for the second thread.
        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 as follows:

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

Passing **this** as the first argument indicates that we want the new thread to call the **run()** method on **this** object. Inside **main()**, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's for loop to begin. Next the main thread 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 varies 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.
```

In a multithreaded program, it is often useful for the main thread to be the last thread to finish running. This program ensures that the main thread finishes last, as the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

**Important note:** Since `NewThread` implements an interface, it can extend another base class if it needs.

### 5.4.2 Extending Thread

We can create a thread by creating a new class that extends **Thread**, and then create an instance of that class. The extending class must override the **run()** method, which is the entry point for the new thread. A call to **start()** begins execution of the new thread. The preceding program is rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {

    NewThread() { // Constructor to create a new, second thread
        super("Demo Thread"); // Invoke Thread constructor
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    public void run() { // This is the entry point for the second thread.
        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. Here the child thread is created by instantiating an object of *NewThread*, which is derived from **Thread**. Also note that the call to **super()** inside *NewThread*, invokes the **Thread** constructor:

```
public Thread(String threadName)
```

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

**Important note:** The *NewThread* class cannot extend any other class, as Java doesn't support multiple inheritance.

### 5.4.3 Choosing an Approach

Which approach among Java's child thread creation is better. The answers is:

- The **Thread** class defines several methods that can be overridden by a derived class. The only method we must override is the **run()** method. This method is also required when we implement **Runnable**.
- Classes should be extended only when they are being enhanced or adapted in some way. Therefore, if we will not be overriding any of **Thread's** other methods, then simply implement **Runnable**. Also, by implementing **Runnable**, our thread class does not need to inherit **Thread**, making it free to inherit a different class.

## 5.5 Creating Multiple Threads

So far, we have been using only two threads: the main thread and one child thread. However, our 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) { // Constructor
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    public void run() { // This is the entry point for thread.
        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 Thread("One"); // start threads
        new Thread("Two");
        new Thread("Three");

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

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

Sample output of 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 we can see here, 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.

## 5.6 Using **isAlive()** and **join()**

Always the main thread has to finish last. This can be accomplished by calling **sleep()** within **main()**, with a long enough delay to ensure that all child threads terminate prior to the main thread. But the question is: How can one thread know when another thread has ended? Fortunately, Thread provides two ways to determine whether a thread has finished. First, we call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is:

**final boolean isAlive()**

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

Second, we more commonly use the **join()** method, to wait for a thread to finish.

**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 us to specify a maximum amount of time that we want to wait for the specified thread to terminate.

The following example 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
    }

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



```
}  
} catch (InterruptedException e) {  
    System.out.println(name + " interrupted.");  
}  
System.out.println(name + " exiting.");  
}  
}  
  
class DemoJoin {  
    public static void main(String args[]) {  
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
        NewThread ob3 = new NewThread("Three");  
  
        System.out.println("Thread One is alive: " + ob1.t.isAlive());  
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());  
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());  
        // wait for threads to finish  
        try {  
            System.out.println("Waiting for threads to finish.");  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Main thread Interrupted");  
        }  
  
        System.out.println("Thread One is alive: " + ob1.t.isAlive());  
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());  
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());  
  
        System.out.println("Main thread exiting.");  
    }  
}
```

Sample output from this program is shown here.

```

New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

```

Here we can see that, after the calls to **join()** return, the threads have stopped executing.

## 5.7 Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower priority thread.

In theory, threads of equal priority should have equal access to the CPU. We need to be cautious due to variations in multitasking implementations across different environments. To ensure safety, threads with the same priority should yield control occasionally, allowing all threads to run, particularly in nonpreemptive operating systems. Although most threads encounter blocking situations, such as waiting for I/O, it is not recommended to rely on this for smooth multithreaded execution. Additionally, for CPU-intensive tasks, it is beneficial to periodically yield control to allow other threads to run.

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

```
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**.

To obtain the current priority setting call the **getPriority()** method of **Thread**.

**final int getPriority()**

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when we 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.

## 5.8 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. Java provides unique, language-level support for it.

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.

Our code can be synchronized in either of two ways, both involving the use of the **synchronized** keyword.

### 5.8.1 Using Synchronized Methods

Synchronization in Java is easy, as 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. 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 need for synchronization:

Consider the following program that 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. Here notice 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 **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 we 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 we 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, we must serialize access to `call()`, i.e: we must restrict its access to only one thread at a time. To do this, we simply need to precede `call()`'s definition with the keyword **synchronized**, as follows:

```

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 we have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, we 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.

### 5.8.2 The synchronized Statement

While creating synchronized methods within classes that we create is an easy and effective means of achieving synchronization, it will not work in all cases. For example, imagine that we 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 us, but by a third party, and we do not have access to the source code. Thus, we 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: We simply put calls to the methods defined by this class inside a **synchronized** block. The general form of the synchronized statement is:

```
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();  
    }  
  
    public void run() { // synchronize calls to call()  
        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");  
  
        try { // wait for threads to end  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```

Here, the `call()` method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside Caller's `run()` method. This causes the same output as the preceding example, because each thread waits for the prior one to finish before proceeding.

## 5.9 Interthread Communication

The previous examples unconditionally blocked other threads from asynchronous access to certain methods. This utilization of implicit monitors in Java objects is powerful, but a more nuanced level of control can be achieved through inter-process communication, which is particularly straightforward in Java.

Multithreading replaces event loop programming by dividing tasks into discrete, logical units. Threads also eliminate the need for polling, which typically involves repeatedly checking a condition within a loop. This approach wastes CPU time. For instance, consider the scenario of a queuing problem where one thread produces data while another consumes it. In a polling system, the consumer would waste CPU cycles waiting for the producer to produce. Once the producer finishes, it would start polling and waste more CPU cycles waiting for the consumer to finish, creating an undesirable situation.

To avoid polling, Java provides an elegant interprocess communication mechanism through the `wait()`, `notify()`, and `notifyAll()` methods. These methods are defined as **final** methods in the **Object** class, making them available to all classes. All three methods can only be called from within a **synchronized** context. Although conceptually advanced, the rules for using these methods are simple:

- **wait()** instructs the calling thread to release the monitor and sleep until another thread enters the same monitor and calls **notify()** or **notifyAll()**.
- **notify()** wakes up a thread that previously called **wait()** on the same object.
- **notifyAll()** wakes up all threads that previously called **wait()** on the same object. One of the threads will be granted access.

These methods are declared in the **Object** class as follows:

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

Additional variations of **wait()** exist that allow specifying a time period to wait.

Before delving into an example illustrating interthread communication, it is important to note that while **wait()** typically waits until **notify()** or **notifyAll()** is called, there is a rare possibility of a spurious wakeup. In such cases, a waiting thread may resume without **notify()** or **notifyAll()** being invoked (essentially resuming for no apparent reason). Due to this remote possibility, the Java API documentation recommends calling **wait()** within a loop that checks the condition on which the thread is waiting. The following example demonstrates this approach.

First, consider the following sample program that incorrectly implements a basic version of the producer/consumer problem. It comprises four classes: **Q** (the synchronized queue), **Producer** (the threaded object responsible for producing queue entries), **Consumer** (the threaded object responsible for consuming queue entries), and **PC** (a small class that creates a single instance of **Q**, **Producer**, and **Consumer**).

*// 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) { // Constructor  
        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) { // Constructor
        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, we 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 we 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:

// 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) { // Constructor  
        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) { // Constructor
        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 **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 **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 **Consumer** that it should now remove it.

Here is the output from this program, which shows the clean synchronous behavior:

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

### 5.9.1 Deadlock

Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects. 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.

Consider 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 calls *deadlockStart()* to start a second thread that sets 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");
    }
    void deadlockStart()
    {
        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[]) {
        Deadlock d1 = new Deadlock();
        d1.deadlockStart();
    }
}
```

In the output shown, execution of *A.foo()* or *B.bar()* will vary based on the specific execution environment.

```
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, we need to press **CTRL-C** to end the program. To see full thread and monitor cache dump press **CTRLBREAK** on the PC. We 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. Therefore, if our multithreaded program locks up occasionally, deadlock is one of the first condition that we should check for.

## 5.10 Suspending, Resuming, and Stopping Threads

Sometimes, it can be useful to suspend the execution of a thread. For instance, a separate thread may be responsible for displaying the current time. If the user decides not to use clock anymore, then the thread associated with it can be suspended. Suspending and resuming a thread is a straightforward process. However, the mechanisms for suspending, stopping, and resuming threads differ between earlier versions of Java (such as Java 1.0) and more modern versions (starting from Java 2). In pre-Java 2 versions, the methods **suspend()**, **resume()**, and **stop()**, defined in the **Thread** class, were used to pause, restart, and stop thread execution. The **suspend()** method was deprecated due to its potential to cause system failures. If a thread is suspended while holding locks on critical data structures, other threads waiting for those resources may become deadlocked.

The **resume()** method is also deprecated because it cannot be used without the **suspend()** method as its counterpart. The **stop()** method was deprecated and also it can lead to system failures. If a thread is stopped while writing to a critically important data structure and has only completed part of its changes, the data structure may end up in a corrupted state. Furthermore, using **stop()** releases any lock held by the calling thread, allowing another thread waiting for the same lock to access the corrupted data.

Although these traditional methods cannot be used, there are alternative approaches to pause, restart, or terminate a thread. The recommended approach involves designing the thread's **run()** method to periodically check a flag variable that indicates the execution state of the thread. This flag can be set to

- "running" to continue thread execution,
- "suspend" to pause the thread, or
- "stop" to terminate it.

By using this approach, the thread can control its own execution.

The example demonstrates how the **wait()** and **notify()** methods inherited from the **Object** class can be used to control thread execution. In the **NewThread** class, there is a **boolean** instance variable called **suspendFlag**, which controls the thread's execution. It is initially set to **false** in the constructor. The **run()** method contains a **synchronized** statement block that checks the **suspendFlag**. If it is **true**, the **wait()** method is invoked to suspend the thread's execution. The **mysuspend()** method sets **suspendFlag** to **true**, while the **myresume()** method sets it to **false** and invokes **notify()** to wake up the thread. Finally, the **main()** method is modified to call the **mysuspend()** and **myresume()** methods.

*// Suspending and resuming a thread for Java 2*

```
class NewThread implements Runnable {  
    String name; // name of thread  
    Thread t;
```

```
boolean suspendFlag;
```

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

```
public void run() { // This is the entry point for thread.
```

```
    try {  
        for(int i = 15; i > 0; i--) {  
            System.out.println(name + ": " + i);  
            Thread.sleep(200);  
            synchronized(this) {  
                while(suspendFlag) {  
                    wait();  
                }  
            }  
        }  
    } catch (InterruptedException e) {  
        System.out.println(name + " interrupted.");  
    }  
    System.out.println(name + " exiting.");  
}
```

```
synchronized void mysuspend() {  
    suspendFlag = true;  
}
```

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

```
class SuspendResume {  
    public static void main(String args[]) {  
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
  
        try {  
            Thread.sleep(1000);  
            ob1.mysuspend();  
        }
```



```

System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.myresume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}

try { // wait for threads to finish
    System.out.println("Waiting for threads to finish.");
    ob1.t.join();
    ob2.t.join();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}

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

```

In this program, the threads suspend and resume.

## 5. 11 Obtaining a Thread's State

A thread can exist in a number of different states. To obtain the current state of a thread call 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**. 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 <b>sleep( )</b> . This state is also entered when a timeout version of <b>wait( )</b> or <b>join( )</b> 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 <b>wait( )</b> or <b>join( )</b> .

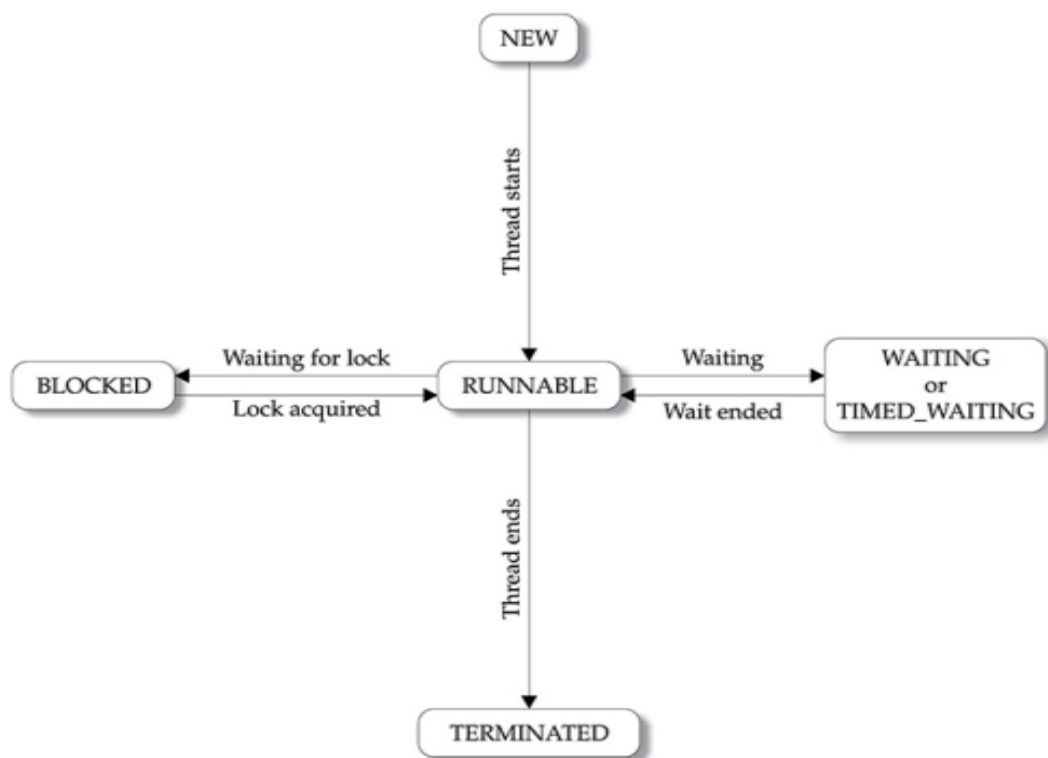


Figure 11-1 Thread states

Given a **Thread** instance, we can use **getState()** to obtain the state of a thread. Example: following code 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) // ...
  
```

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. Therefore, **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.

## 5.12 Enumerations:

### What is enumeration?

Java versions prior to JDK 5 lacked one feature that many programmers felt was needed: that is enumerations. An enumeration in its simplest form, is a list of named constants. In Java, an enumeration defines a class type. Also, an enumeration can have constructors, methods, and instance variables.

### When to use enumeration?

Sometimes we may want a variable that can take on only a certain listed (enumerated) set of values

Examples:

- daysOfAWeek: SUNDAY, MONDAY, TUESDAY, ...
- monthsOfAYear: JANUARY, FEBRUARY, MARCH, APRIL, ...
- gender: MALE, FEMALE
- title: MR, MRS, MS, PROF, DR
- appletState: READY, RUNNING, BLOCKED, DEAD

The values are written in all caps because they are constants.

What is the actual type of these constants?

Except for constructors, an Enum is an ordinary class. Each name listed within an Enum is actually a call to a constructor

Example 1:

- **enum** Season { WINTER, SPRING, SUMMER, FALL }

This constructs the four named objects, using the default constructor

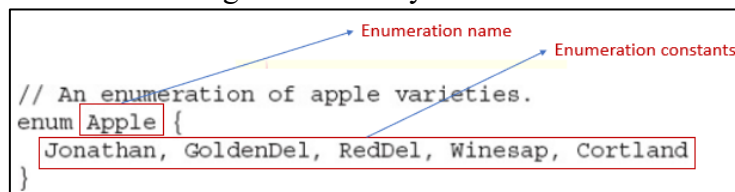
Example 2:

```
public enum Coin
{
    private final int value;
    Coin(int value)
    {
        this.value = value;
    }
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
}
```

Enum constructors are only available within the Enum itself. An enumeration is supposed to be complete and unchangeable.

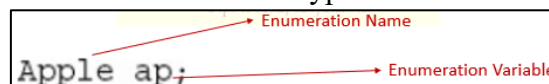
### 5.12.1 Enumeration Fundamentals

- An enumeration is created using the **enum** keyword.



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

- We can create a variable of enumeration type as follows:



```
Apple ap;
```

- Enumeration variable can be assigned or can contain only those values that are defined by the enumeration.

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

- Two enumeration constants can also be compared for equality by using the == relational operator.

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

- An enumeration value can also be used to control a switch statement if all the case statements use same enum constants as that used by the switch expression.

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

- When an enumeration constant is displayed, such as in a `println()` statement, its name is output. (*//displays Winesap.*)

```
System.out.println(Apple.Winesap);
```

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

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

        ap = Apple.RedDel;

        // Output an enum value.
        System.out.println("Value of ap: " + ap);
        System.out.println();

        ap = Apple.GoldenDel;

        // Compare two enum values.
        if(ap == Apple.GoldenDel)
            System.out.println("ap contains GoldenDel.\n");

        // Use an enum to control a switch statement.
        switch(ap) {
            case Jonathan:
                System.out.println("Jonathan is red.");
                break;
            case GoldenDel:
                System.out.println("Golden Delicious is yellow.");
                break;
            case RedDel:
                System.out.println("Red Delicious is red.");
                break;
            case Winesap:
                System.out.println("Winesap is red.");
                break;
            case Cortland:
                System.out.println("Cortland is red.");
                break;
        }
    }
}
```

**Output:**

```
Value of ap: RedDel
ap contains GoldenDel.
Golden Delicious is yellow.
```

**5.12.2 The values() and valueOf() Methods**

- All enumerations contain two predefined methods:
- **values()** and **valueOf()**.
- General format:  
**public static enum-type [ ] values()**  
**public static enum-type valueOf(String str)**
- **values()** method: returns an *array list of the enumeration constants*.
- **valueOf()** method: returns the *enumeration constant* whose value corresponds to the string passed in **str**.

**Apple.valueOf("Winesap")** //returns Winesap

```
// 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:");
        Apple allapples[] = Apple.values(); // use values()
        for(Apple a : allapples)
            System.out.println(a);
        System.out.println();
        ap = Apple.valueOf("Winesap"); // use valueOf()
        System.out.println("ap contains " + ap);
    }
}
```

**Output:**

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

ap contains Winesap
```

**5.13 Type wrappers**

Java uses primitive types (or simple types), such as **int** or **double**, to hold the basic data types supported by the language. But primitive types cannot be passed by reference to a method. Also, as Java standard data structures operate on objects, these data structures cannot be used 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. These classes offer a wide array of methods that allow us to fully integrate the primitive types into Java's object hierarchy.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**.

- **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( )**, as **char charValue( )** It returns the encapsulated character.

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

**Boolean(boolean boolValue)**

Here, *boolValue* must be either *true* or *false*.

**Boolean(String boolString)**

Here, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be **true**. Otherwise, it will be **false**. Use **booleanValue( )**, to obtain a **boolean** value from a **Boolean** object.

**boolean booleanValue( )** //It returns the boolean equivalent of the invoking object.

- **The Numeric Type Wrappers**

- All the numeric type wrappers - **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double** - inherit the abstract class **Number**.

- **Number** declares the following methods that return the value of an object in different number formats:

**byte byteValue( )**

**double doubleValue( )**

**float floatValue( )**

**int intValue( )**

**long longValue( )**

**short shortValue( )**

- 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.
- Example, the constructors defined for **Integer** are:

**Integer(int num)**

**Integer(String str)**

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

- All the type wrappers override **toString( )** and returns the value contained within the wrapper, which allows the user to output the value by passing a type wrapper object to **println( )**, without having to convert it into its primitive type.

```
// Demonstrate a type wrapper.
class Wrap {
    public static void main(String args[]) {
        Integer iOb = new Integer(100); // Boxing
        int i = iOb.intValue(); // Unboxing
        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

## 5.14 Autoboxing

- Two important features of Java's JDK 5 version:
  - autoboxing and auto-unboxing.
- Autoboxing** - process by which a primitive type is automatically encapsulated or boxed into its equivalent type wrapper whenever an object of that type is needed.
- Auto-unboxing** - process by which the value of a boxed object is automatically extracted or unboxed from a type wrapper when its value is needed.
- Autoboxing and auto-unboxing:
  - Greatly streamlines the coding of several algorithms,
  - Removes the tedium of manually boxing and unboxing values.
  - Helps in preventing errors.
  - Very important to generics, which operate only on objects.
  - Autoboxing makes working with the Collections Framework much easier.
- With autoboxing, we just need to only assign that value to a type-wrapper reference and Java automatically constructs the object for us.

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

- Note:** The object is not explicitly created using of **new**. Java automatically handles this.
- To unbox an object, assign that object reference to a primitive-type variable.

Example, to unbox iOb: `int i = iOb; // auto-unbox`

```
// 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
    }
}
```

### 5.14.1 Autoboxing and Methods

- 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. For example:

```
// 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);
    }
}
```

In the program, notice that method **m()** specifies an **Integer** parameter and returns an **int** result. Inside **main()**, **m()** is passed the value **100**. Because **m()** is expecting an **Integer**, this value is automatically *boxed*. Then, **m()** returns the **int** equivalent of its argument. This causes **v** to be *auto-unboxed*. Next, this **int** value is assigned to **iOb** in **main()**, which causes the **int** return value to be *autoboxed*.

#### 5.14.2 Autoboxing/Unboxing Occurs in Expressions

- Autoboxing and unboxing are applicable to expressions too.
- In an expression, a numeric object is automatically unboxed and the outcome of the expression is reboxed, if at all it is necessary.

```
// 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);
    }
}
```

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

In this program observe the line `++iOb`. This is a pre-increment expression which causes the value in **Integer** object **iOb** to be incremented. First the value within the **Integer** object **iOb** is unboxed, then the value is incremented, and lastly the resulting value is then reboxed.

- Auto-unboxing allows different types of numeric objects to be mixed in an expression.
- After auto-unboxing, the standard type promotions and conversions are applied.

```
class AutoBox4 {  
    public static void main(String args[]) {  
        Integer iOb = 100;  
        Double dOb = 98.6;  
  
        dOb = dOb + iOb;  
        System.out.println("dOb after expression: " + dOb);  
    }  
}
```

```
dOb after expression: 198.6
```

Here, both the **Double** object **dOb** and the **Integer** object **iOb** participates in the addition, and the result is reboxed and stored back into **dOb**.

- Auto-unboxing, use Integer numeric objects to control a switch statement.

```
Integer iOb = 2;  
  
switch(iOb) {  
    case 1: System.out.println("one");  
        break;  
    case 2: System.out.println("two");  
        break;  
    default: System.out.println("error");  
}
```

When the switch expression is evaluated, **iOb** is unboxed and its int value is obtained, which is later compared against each of the case labels within the switch block.

### 5.14.3 Autoboxing/Unboxing Boolean and Character Values

- Autoboxing/unboxing can also be applied on wrappers of boolean and char - **Boolean** and **Character** respectively.
- Example:

```
// 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);
    }
}
```

**Output:**

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

The most important thing to be noticed in this program is the auto-unboxing of **b** inside the **if** conditional expression. We know that in a conditional expression that controls an **if**, it must evaluate to a value of type **boolean**. Because of auto-unboxing, the **boolean** value contained within **b** is automatically unboxed when the conditional expression is evaluated.

- With the advent of autoboxing/unboxing, a **Boolean** object can be used to control an **if** statement and also any of Java's loop statements.
- When a Boolean is used as the conditional expression of a **while**, **for**, or **do..while**, it automatically unboxes into its **boolean** equivalent.
- Example:

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