UNIT VI

Multithreading - Differences between multi threading and multitasking, thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication, thread groups, daemon threads.
Enumerations, autoboxing, annotations, generics..

Executing the tasks is of two types:

· **Single Tasking**: Executing only one task at a time is called single tasking. In this single tasking the microprocessor will be sitting idle for most of the time. This means micro processor time is wasted.
· **Multi tasking**: Executing more than one task at a time is called multi tasking.

Multitasking is of two types:

o **Process Based Multitasking:** Executing several programs simultaneously is called process based multi tasking.
o **Thread Based Multitasking**: Executing different parts of the same program simultaneously with the help of a thread is called thread based multitasking.

Advantage of multitasking is utilizing the processor time in an optimum way.

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. Thread is a smallest unit of code. Thread is also defined as a sub process. A Thread sometimes called an execution context or a light weight process.

DIFFERENCES BETWEEN MULTITHREADING AND MULTITASKING

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.

There are two distinct types of multitasking: process-based and thread-based.
A *process* is a program that is executing. Thus, *process-based* multitasking is the feature that allows 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.
In **process-based** multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
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 task s simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

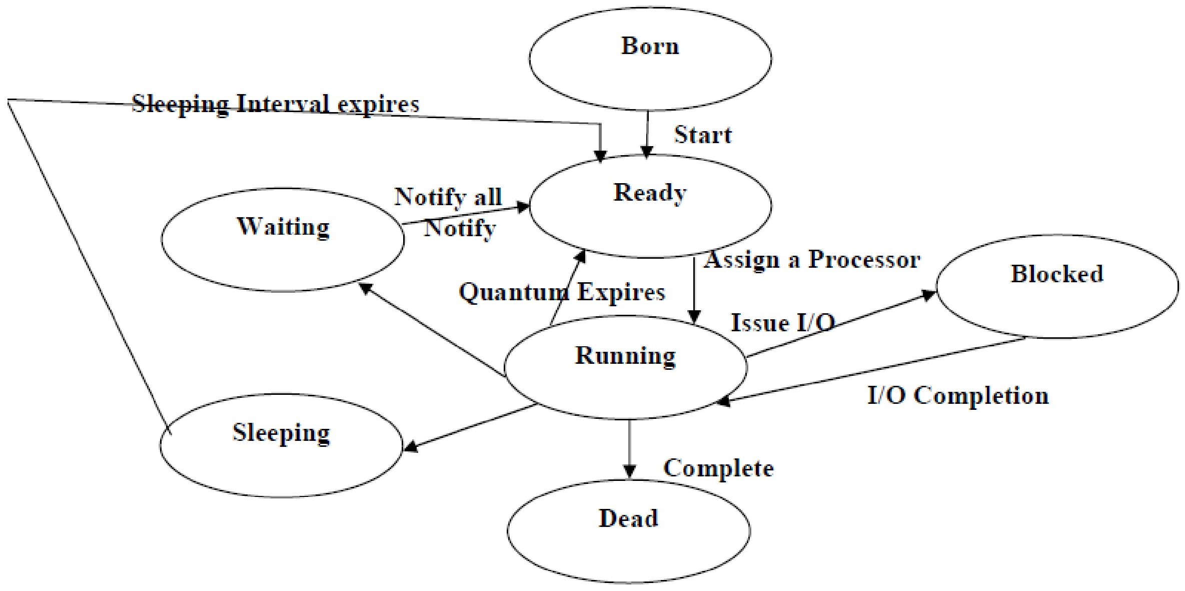
Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.

Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.

THREAD LIFE CYCLE:

Thread States (Life-Cycle of a Thread): The life cycle of a thread contains several states. At any time the thread falls into any one of the states.



- ✓ The thread that was just created is in the born state.
- ✓ The thread remains in this state until the threads start method is called. This causes the thread to enter the ready state.
- ✓ The highest priority ready thread enters the running state when system assigns a processor to the thread i.e., the thread begins executing.
- ✓ When a running thread calls wait the thread enters into a waiting state for the particular object on which wait was called. Every thread in the waiting state for a

given object becomes ready on a call to notify all by another thread associated with that object.

- ✓ When a sleep method is called in a running thread that thread enters into the suspended (sleep) state. A sleeping thread becomes ready after the designated sleep time expires. A sleeping thread cannot use a processor even if one is available.
- ✓ A thread enters the dead state when its run () method completes (or) terminates for any reason. A dead thread is eventually be disposed of by the system.
- ✓ One common way for a running thread to enter the blocked state is when the thread issues an input or output request. In this case a blocked thread becomes ready when the input or output waits for completes. A blocked thread can't use a processor even if one is available.

Uses of Threads:

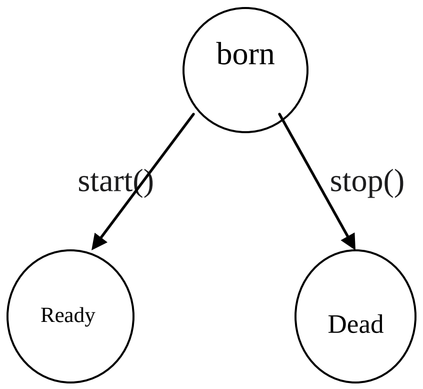
· Threads are used in designing server side programs to handle multiple clients at a time.
· Threads are used in games and animations.

Born State

When we create a thread object, the thread is born and is said to be in new born state. The thread is not yet scheduled for running. At this state, we can do only one of the following:

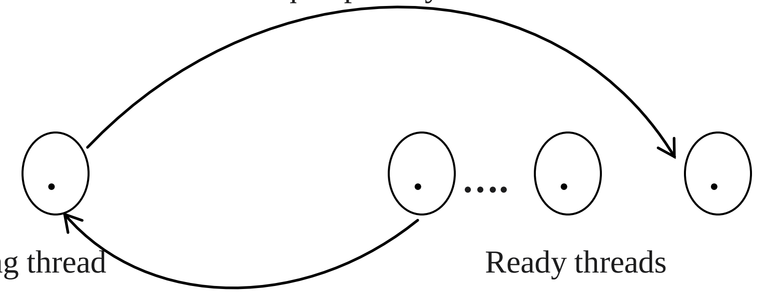
- Scheduled it for running using start() method.
- Kill it using stop() method.

If scheduled it moves to the ready state. If we attempt to use any other method at this stage, an exception will be thrown.



Ready State

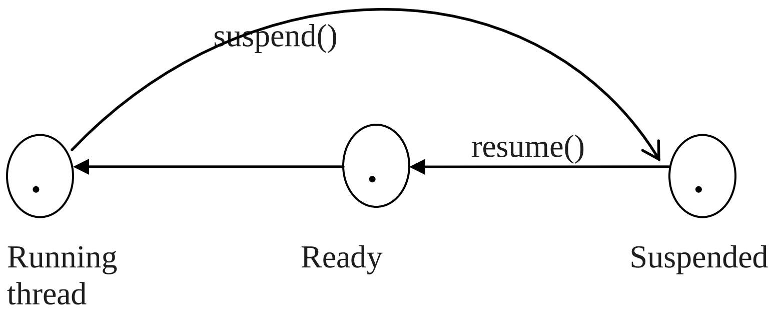
The ready state means that the thread is ready for execution and is waiting for the availability of the processor. i.e. the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for executin in round robin fashion. i.e first come first serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as time slicing. If we want a thread to relinquish control to another thread of equal priority before its turn comes.



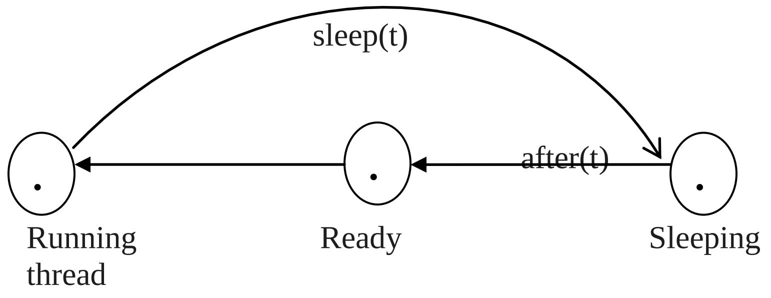
Running State

Running means that the processor has given its time to the thread for its execution. The thread rubs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

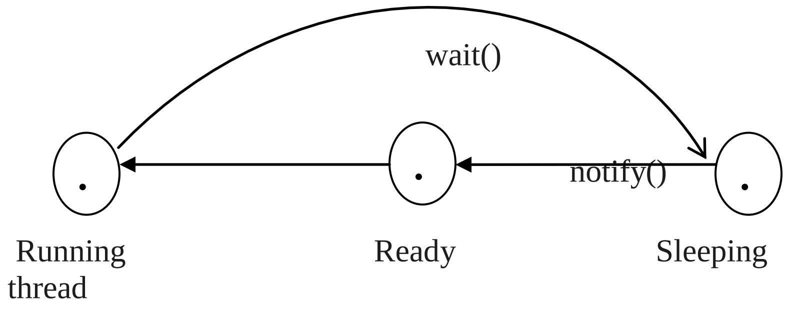
1. It has been suspended using suspend() methos. A suspended thread can be revived by using resume() method. this approach is useful when we want to suspend a thread for some time due to certaing reasons, but do not want to kill it.



2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method sleep(time) where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.



3. It has been told to wait until some event occurs. This is done using the wait() method. The thread can be scheduled to run again using the notify() method.



Blocked State

A thread is said to be blocked when it is prevented from entering into the ready state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain conditions. A blocked thread is considered “not ready” but not dead and therefore fully qualified to run again.

Dead State

Every thread has a life cycle. A running thread ends its life when it has completed executing its run() method. It is a natural death. We can kill it by sending the stop() message to it at any state thus causing a premature death to it. A thread can be killed as soon as it is born, or while it is running, or when it is in blocked state.

CREATING THREADS

Java’s multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

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

The **Thread** class defines several methods that help manage threads.

Method	**Meaning**
getName	Obtain a thread’s name.
getPriority	Obtain a thread’s priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Creating a Thread:

- ✓ Write a class that extends Thread class or implements Runnable interface this is available in lang package.
- ✓ Write public void run () method in that class. This is the method by default executed by any thread.
- ✓ Create an object to that class.
- ✓ Create a thread and attach it to the object.
- ✓ Start running the threads.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is called as *main thread*, because it is the one that is executed when the program begins.

The main thread is important for two reasons:
■ It is the thread from which other "child" threads will be spawned.
■ Often it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when program is started, it can be controlled through a **Thread** object. To do so, we must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.

Its general form is shown here:

static Thread currentThread( )

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

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

After creating a class that implements **Runnable**, we 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*)

After the new thread is created, it will not start running until we call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**.

```
void start( )
```

Example:

```
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
```

```
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
}
catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}

class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Output:

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

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.

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

CREATING MULTIPLE THREADS

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

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

class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
```

```
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

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

final void setPriority(int *level*)

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

W can obtain the current priority setting by calling the **getPriority( )** method of **Thread**,
final int getPriority( )

Thread priorities can change from 1 to 10. We can also use the following constants to represent priorities:

Thread.MAX_PRIORITY value is 10
Thread.MIN_PRIORITY value is 1
Thread.NORM_PRIORITY value is 5

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

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

SYNCHRONIZATION

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.*

Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

· To synchronize an entire method code we can use synchronized word before method name
e Thread synchronization is done in two ways:
· Using synchronized block we can synchronize a block of statements.
e.g.: synchronized (obj)
{
statements;
}**.g.:** synchronized void method ()
{
}

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. 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.

```
// 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]
]
]
```

serialize access to call( ). i.e, restrict its access to only one thread at a time. To do this, simply 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]
```

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(object) {
// statements to be 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();
}
// 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

- ✓ 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. 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 interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are

implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

■ **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.

■ **notify( )** wakes up the first thread that called **wait( )** on the same object.

■ **notifyAll( )** wakes up all the threads that called **wait( )** on the same object.

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

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

```
// 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
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if(!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) {
if(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
```

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

Inside **get( )**, **wait( )** is called. This causes its execution to suspend until the **Producer** notifies you that some data is ready. When this happens, execution inside **get( )** resumes. After the data has been obtained, **get( )** calls **notify( )**. This tells **Producer** that it is okay to put more data in the queue. Inside **put( )**, **wait( )** suspends execution until the **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called. This tells the **Consumer** that it should now remove it.

Here is some output from this program, 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
```

THREAD GROUP

A ThreadGroup represents a group of threads. The main advantage of taking several threads as a group is that by using a single method, we will be able to control all the threads in the group.

· Creating a thread group: ThreadGroup tg = new ThreadGroup (“groupname”);

· To add a thread to this group (tg): Thread t1 = new Thread (tg, targetobj, “threadname”);

· To add another thread group to this group (tg):
ThreadGroup tg1 = new ThreadGroup (tg, “groupname”);

· To know the parent of a thread: tg.getParent ();

· To know the parent thread group: t.getThreadGroup (); This returns a ThreadGroup object to which the thread t belongs.

· To know the number of threads actively running in a thread group: t.activeCount ();

· To change the maximum priority of a thread group tg: tg.setMaxPriority ();

- ✓ Thread and ThreadGroup are the classes present in java.lang package.
- ✓ ThreadGroup creates a group of threads.
- ✓ It is possible to suspend all threads at a time with ThreadGroup.

Two constructors in ThreadGroup are:

ThreadGroup(String groupName)
ThreadGroup(ThreadGroup ob, String groupName)

```
class ThreadGroupDemo
{
public static void main(String args[])
{
Thread t=new Thread();
ThreadGroup tg=new ThreadGroup("Group1");
Thread1 t1=new Thread1(tg, "ChildThread1");
Thread2 t2=new Thread1(tg, "ChildThread2");
t1.start();
t2.start();
System.out.println("No of threads in group are.."+tg.activeCount());
}
}

class Thread1 extends Thread
{
Thread1(ThreadGroup tg, String name)
{
super(tg.name);
}
public void run()
{
try
{
for(int i=1;i<=2;i++)
{
System.out.println("From child thread1");
Thread.sleep(1000);
}
}
catch(InterruptedException e)
{
System.out.println(e);
}
}
}

class Thread2 extends Thread
{
Thread1(ThreadGroup tg, String name)
{
super(tg,name);
}
public void run()
{
try
{
for(int i=1;i<=2;i++)
{
System.out.println("From child thread2");
Thread.sleep(1000);
}
}
catch(InterruptedException e)
{
System.out.println(e);
}
}
}
```

```
No of threads in Group1: 2
From childthread 1
From childthread 2
From childthread 1
From childthread 2
```

DAEMON THREAD

- ✓ In Java, any thread can be a Daemon thread.
- ✓ Daemon threads are like a **service providers** for other threads or objects running in the same process as the daemon thread.
- ✓ Daemon threads are used for background supporting tasks and are only needed while normal threads are executing.
- ✓ If normal threads are not running and remaining threads are daemon threads then the interpreter exits.

setDaemon(true/false) ? This method is used to specify that a thread is daemon thread.

public boolean isDaemon() ? This method is used to determine the thread is daemon thread or not.

The core difference between user threads and daemon threads is that the JVM will only shut down a program when all user threads have terminated. Daemon threads are terminated by the JVM when there are no longer any user threads running, including the main thread of execution. Use daemons as the minions they are.

```
public class DaemonThread extends Thread
{
public void run()
{
System.out.println("Enter run method");
try
{
System.out.println("In run method: currentThread() is" +Thread.currentThread());

while(true)
{
try
{
Thread.sleep(500);
}
catch(InterruptedException x)
{
}

System.out.println("In run method: woke up again");
}
}

public static void main(String args[])
{
System.out.println("Entering main method");

DaemonThread t=new DaemonThread();
t.setDaemon(true);
t.start();

try
{
Thread.sleep(3000);
}
catch(InterruptedException x)
{
}

System.out.println("Leaving main thread");
}
}
```

ENUMERATIONS

An enumeration is a list of named constant.
In C++ enumerations are simply list of named integer constants. In Java an enumeration defines a class type.

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

```
enum Apple
{
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

The identifiers Jonathan, GoldenDel ansd so on.. are called enumeration constants. Each is implicitly declared as public and static.

Once an enumeration is defined, we can create a variable of that type.
Apple ap;

We cannot instatntiate an enum using new.

Eg:

```
enum Apple
{
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

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

ap=Apple.RedDel;

System.out.println("Value of ap:" +ap);
System.out.println();
ap=Apple.GoldenDel;

if(ap==Apple.GoldenDel)
System.out.println("ap contains goldenDel");

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;
} } }
```

```
Value of ap: RedDel
ap contains GoldenDel
Golden Deliciou is yellow
```

The values() ans valueOf() Methods

All enumerations automatically contain two predefined methods: values() and valueOf()

```
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 returhs the enumeration constant whose value corresponds to the string passed in str.

```
Apple allapples[] = Apple.values();
```

```
for (Apple a : allapples)
System.out.println(a)
```

or

```
for(Apple a : Apple.values())
System.out.println(a);


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

Java Enumerations Are Class Types

A Java enumeration is a class type, although we don't instantiate an enum using new.
Each enumeration constant is an object of its enumeration type.
When a constructor for an enum is defined, it is called when each enumeration constant is created.
Each enumeration constant has its own copy of any instance variables defined by the enumeration.

```
enum Apple
{
Jonathan(10, GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

private int price;

Apple(int p)
{
price = p;
}

int getPrice()
{
return price;
}
}

class EnumDemo
{
public static void main(String args[])
{
Apple ap;
System.out.println("Winesap costs" + Apple.Winesap.getPrice() + "cents");
System.out.println("All apple prices:");
for(Apple a : Apple.values())
System.out.println(a+ "costs" +a.getPrice() + "cents");
}
}
```

```
Winesap costs 15 cents.

All apple prices:
Jonathan costas 10 cents
GoldenDel costs 9 cents
RedDel costs 12 cents
Winesap costs 15 cents
Cortland costs 8 cents.
```

Enumerations Inherit Enum

All enumerations automatically inherit java.lang.Enum. This class defines several methods that are available for use by all enumerations.

We can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal vale.

final int ordinal()

We can compare the ordinal value of two constants of the same enumeration using compareTo().

final int compareTo(enum-type e)

We can compare for equality an enumeration constant wikth any other object by using **equals().**

AUTOBOXING

Autoboxing is the process by which a primitive type is automatically encapsulated 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 from a type wrapper when its value is needed. There is no need to call a method such as intValue() or doubleValue().

```
Integer i = 100;
```

To unbox an object: int j = i;

Autoboxing and Methods

Autoboxing automatically occurs whenever a primitive type must be converted into an object; Auto-nboxing 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.

```
class Autoboxing
{
static int meth(Integer v)
{
return v;
}

public static void main(String args[])
{
Integer ib = meth(100);
System.out.println(ib);
}
}
```

Autoboxing/Unboxing Boolean and Character Values

Java supplies wrapper classes for Boolean and char, thease are Boolean and Character. Autoboxing/Unboxing applies to these wrappers.

```
class AutoBox
{
public static void main(String args[])
{
Booleab b=true;

if(b)  // b is auto unboxed
System.out.println("b is true");

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

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

METADATA (ANNOTATIONS)

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compiler-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at runtime.

Annotations can be applied to a program's declarations of classes, fields, methods, and other program elements.

The annotation appears first, often (by convention) on its own line, and may include *elements* with named or unnamed values:

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
```

```
@interface MyAnno{
String str();
int val();
}
```

- ✓ @ that prcedes the keyword interface tells the compiler that an annotation type is being declared. We don’t provide bodies for these methods, Java implements these methods.
- ✓ An innotation cannot include an extends clause.
- ✓ All annotation types automatically extend the Annotation interface. Annotation is a super interface of all annotations. It is declared within the java.lang.annotation package.

Specifying a Retention Policy

A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within java.lang.annotation.RetentionPolicy enumeration. They are SOURCE, CLASS and RUNTIME.

An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of CLASS is stored in the .class file during compilation. It is not available through the JVM during run time.

An annotation with a retention policy of RUNTIME is stored in the .class file during the compilation and is available through the JVM during run time.

```
@Retention(retention-policy)
```

Eg:

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno
{
String str();
int val();
}
```

Many annotations replace what would otherwise have been comments in code.

Suppose that a software group has traditionally begun the body of every class with comments providing important information:

```
public class Generation3List extends Generation2List {

   // Author: John Doe
   // Date: 3/17/2002
   // Current revision: 6
   // Last modified: 4/12/2004
   // By: Jane Doe
   // Reviewers: Alice, Bill, Cindy

   // class code goes here

}
```

To add this same metadata with an annotation, you must first define the *annotation type*. The syntax for doing this is:

```
@interface ClassPreamble {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
```

```
  String lastModifiedBy() default "N/A";
  // Note use of array
  String[] reviewers();
}
```

GENERICS

JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of *generics*.
Using generics it is possible to create a single class that automatically works with different types of data.

General form of a Generic class:

Declaring a generic class

```
class class-name<type-param-list>
{

}
```

Declaring a reference to a generic class:

```
class class-name<type-arg-list> var-name = new class-name<type-arg-list>(cons-arg-list);
```

```
class Gen<T>
{
T ob;

Gen(T o)
{
ob=o;
}

T getob()
{
return ob;
}

void showType()
{
System.out.println(ob.getClass().getName());
}
}

class GenDemo
{
public static void main(String args[])
{
Gen<Integer> ib = new Gen<Integer>(88);

ib.showType();

int v=ib.getob();
System.out.println(v);

Gen<String> strob=new Gen<String>(“Generics Test”);
strob.showType();

String str=strob.getob();
System.out.println(str);
}
}
```

Output:

```
java.lang.Integer
88
java.lang.String
Generics Test
```

A Generic Class with Two Type Parameters

```
class TwoGen<T, V>
{
T ob1;
V ob2;

TwoGen(T o1, V o2)
{
ob1=o1;
ob2=o2;
}

void showTypes()
{
System.out.println(“Type of T is” +ob1.getClass().getName());
System.out.println(“Type of V is” +ob2.getClass().getName());
}
T getob1()
return ob1;
}

V getob2()
{
return ob2;
}
}

class SimpGen
{
public static void main(String args[])
{
TwoGen<Integer, String> obj = new TwoGen<Integer, String> (88, “Generics”);

obj.showTypes();

int v=obj.getob1();
System.out.println(v);

String str=obj.getob2();
System.out.println(str);
}
}
```

```
Type of T is java.lang.Integer
Type of V is java.lang.String
88
Generics
```