→→ **Multi-Threading: -** Java provides built-in support for *multithreaded programming.* A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. However, there are two distinct types of multitasking: process-based and thread-based.

- A *process* is a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. 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 i.e. that a single program can perform two or more tasks 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. Inter-process 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. Inter-thread communication is inexpensive, and context switching from one thread to the next is low cost.

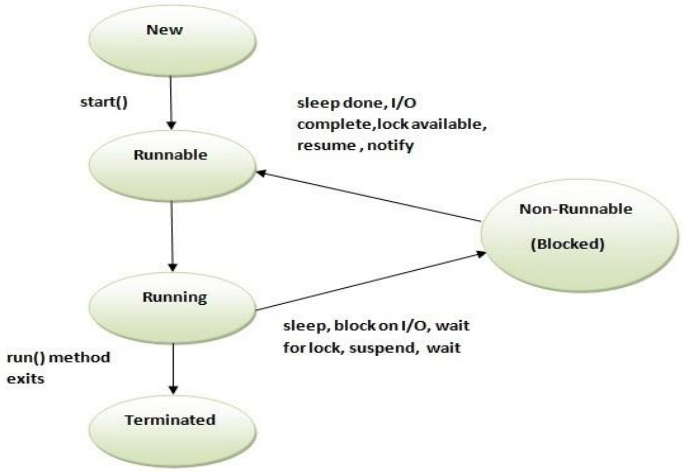
→ Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

→ **Differences between multi-threading and multitasking:**

MULTI THREADING	MULTI TASKING
More than one thread running simultaneously	More than one process running simultaneously
It's a part of program	It's a program.
It is a light-weight process.	It is a heavy-weight process.
Threads are divided into sub threads	Process is divided into threads.
Within the process threads are communicated.	Inter process communication is difficulty
Context switching between threads is cheaper.	Context switching between process is costly
It is controlled by Java(JVM)	It is controlled by operating System.
It is a specialized form of multitasking	It is a generalized form of multithreading.

→→ **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 these states.

- New thread
- Runnable
- Running
- Blocked
- Dead



- New Thread: When a thread is created it is in a new state. In this state system resources are not allocated to the thread.
- Runnable: When the start() method is called on the thread object, the thread is in runnable state. A runnable thread is ready for execution, but is not executed currently. The runnable thread takes the CPU determined by the OS and goes to running state.
- Running: A thread being executed by the CPU is in a running state.
- Blocked: A running thread may go to a blocked state due to any of the following conditions:
  - A wait method is called by the thread.

- The thread performs I/O operations
- A sleep method is called by the thread.
- A suspend method is called by the thread.

A thread which has moved to blocked state goes into the runnable state by the following ways:

- If a thread has been put to sleep() the specified timeout period must expire
- If a thread has called wait() then some other thread using the resource for which the first thread is waiting must call notify() or notifyAll()
- If a thread is waiting for the completion of an input or output operation, then the operation must finish.
- If a thread has called suspend() then some other thread using the resource for which the thread is suspended must call resume().

When the blocked thread is unblocked, it goes to runnable state and not to running state.

- Dead: A thread goes into dead state in two ways.
  - If its run() exits: a thread completes its task.
  - A stop() is invoked: this method kills the thread.

→→ Creating a Thread:

A thread can be created by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

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

→ Implementing Runnable: The easiest way to create a thread is to create a class that implements the **Runnable** interface.

Runnable abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this: **public void run( )**

Inside **run( )**, define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only

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

After creating a class that implements **Runnable**, instantiate an object of type **Thread** from within that class. **Thread** defines several constructors.

Thread(Runnable *threadOb*, String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

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

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ThreadDemo {
public static void main(String args[]) {
```

```
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement: t = new Thread(this, "Demo Thread"); Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object. Next, **start( )** is called, which starts the thread execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin. After calling **start( )**, **NewThread**'s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows.

```
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.
```

Notice the output produced when **t** is used as an argument to **println( )**. This display, in order: the name of the thread, its priority, and the name of its group. The name of the child thread is Demo Thread. Its priority is 5, which is the default value, and **main** is the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole.

→ **Extending Thread:** The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

This program generates the same output as the preceding version. As we can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**. Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)

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

What is the difference between 'extends thread' and 'implements Runnable' ? which one is advantageous?

'extends Thread' and 'implements Runnable'-both are functionally same. But when we write extends Thread, there is no scope to extend another class, as multiple inheritance is not supported in java.

→→ **Creating Multiple Threads:** Multiple threads is a concept of creating multiple threads. Our program can create as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
```

```
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}
}
class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
```

```
}
The output from this program is shown here:
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main( )**.

→ **The Main Thread:** When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

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

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

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

→ **Using isAlive( ) and join( ):** As mentioned earlier we want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread. How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here: *final boolean isAlive( )*

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

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

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
}
class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "+
ob1.t.isAlive());
System.out.println("Thread Two is alive: "+
ob2.t.isAlive());
System.out.println("Thread Three is alive: "+
ob3.t.isAlive());
// wait for threads to finish
try {
System.out.println("Waiting for threads to
finish.");
ob1.t.join();
ob2.t.join();
```

```
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here. (output may vary based on processor speed and task load.)

```
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.
```

→ **Multiple Threads acting on a single resource:** If several threads act on a single resource which may lead to confusion. The following program illustrates about the problem.

```
// This program is not synchronized.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run() {
target.call(msg);
}
}
class Synch {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
```

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

```
}
}
}
Here is the output produced by this program:
Hello[Synchronized[World]
]
]
```

As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition,* because the three threads are racing each other to complete the method.

To avoid methods serving many threads to leave in the middle before completing the task, java provides a mechanism called 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. To achieve the synchronization in java a keyword called **synchronized** is used.

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

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

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

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

The synchronized Statement: The following is the general form of the **synchronized** statement:

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

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor. Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

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

```
// This program uses a synchronized block.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
```

```
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target,
"Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
```

```
}
```

```
}
```

→→ **Inter Thread Communication:** Threads are created to carry out light-weight process independently. In certain problems, two or more threads may use an object as a common resource. In order to avoid mix-up of the task of one thread with that of another thread, the resource object is synchronized. When one thread is using the synchronized object, the monitor is locked and another thread needing to use this object has to keep waiting. A synchronized object may have more than one synchronized method. One thread may need to use one synchronized method, while another thread may need another synchronized method of the same object. But when a synchronized object is used by one thread, it cannot be accessed by any other thread, even if a different method of the shared object is needed. It may happen that only after an action has taken place in one thread, the other thread can proceed. If the currently running thread can proceed only after an action in another non-running thread, the running thread has to keep waiting infinitely. To avoid such problem, java provides inter-thread communication methods, which can send messages from one thread to another thread, which uses the same object. The methods used for inter-thread communication are:

- wait() : This method makes the calling thread to give up the monitor and go to sleep until some other thread wakes it up.
- notify(): This method wakes up the first thread which called wait() on the same object.
- notifyAll():This method wakes up all the threads that called wait() on the same object.

All the three methods can be called only inside a synchronized code and are applicable to threads that share the same object.

```
// A correct implementation of a producer and consumer problem:
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
while(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
```

```
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
while(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
```

```
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PCFixed {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
```

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

```
Put: 1
Got: 1
Put: 2
Got: 2
```

→→ **Deamon Threads:** Daemon threads are sometimes called "service" threads that normally run at a low priority and provide a basic service to a program or programs when activity on a machine is reduced. An example of a daemon thread that is continuously running is the garbage collector thread. This thread, provided by the JVM, will scan programs for variables that will never be accessed again and free up their resources back to the system.

A deamon thread is a thread that executes continuously. Daemon threads are service providers for other threads or objects. It generally provides a background processing.

- To make a thread t as a deamon thread, we can use setDeamon() method as: t.setDeamon(true);
- To know if a thread is deamon or not, isDeamon is useful: boolean x=t.isDaemon().

Write a example program for setting a thread as a daemon thread

```
public class DaemonDemo extends Thread
{
public void run()
{
for(int i=0;i<5;i++)
System.out.println(this.getName()+" :"+i);
}
public static void main(String args[])
{
DaemonDemo d1=new DaemonDemo();
DaemonDemo d2=new DaemonDemo();
d1.setName("Daemon thread");
d2.setName("Normal thread");
d1.setDaemon(true);
```

```
d1.setPriority(Thread.MIN_PRIORITY);
d1.start();
d2.start();
}
}
Output:
Daemon Thread : 0
Normal Thread : 0
```

```
Daemon Thread : 1
Normal Thread : 1
Daemon Thread : 2
Normal Thread : 2
Daemon Thread : 3
Normal Thread : 3
Daemon Thread : 4
Normal Thread : 4
```

→→ **Thread Groups:** A thread group represents several threads as a single group. 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.

ThreadGroup provides the following two constructors:

- public ThreadGroup(String TGName) constructs a ThreadGroup with TGName
- public ThreadGroup(ThreadGroup ParentTG, String TGName) constructs a child ThreadGroup of parentTG called TGName.

The class Thread provides constructors that enable the programmer to instantiate a Thread and associate it with a ThreadGroup.

- public Thread(ThreadGroup TG, String TName) constructs a Thread that belongs to TG and has the name TName. This constructor is normally invoked for derived classes of Thread.
- public Thread(ThreadGroup TG, Runnable RObject, String TName) constructs a Thread that belongs to TG and has the name TName. This constructor is normally invoked for implementing Runnable Interface.

Program to demonstrate the creation of ThreadGroups and some methods which acts on ThreadGroups?

```
class Reservation implements Runnable
{
public void run()
{
System.out.println("I am Reservation Thread");
}
}
```

```
class Cancellation implements Runnable
{
public void run()
{
System.out.println("I am Cancellation Thread");
}
}
class TGroups
{
```

```
public static void main(String[] args) throws Exception
{
Reservation res=new Reservation();
Cancellation can=new Cancellation();
ThreadGroup tg=new ThreadGroup("First Group");

Thread t1=new Thread(tg,res,"First Thread");
Thread t2=new Thread(tg,res,"Second Thread");
ThreadGroup tg1=new ThreadGroup(tg,"Second Group");
Thread t3=new Thread(tg1,can,"Third Thread");
Thread t4=new Thread(tg1,can,"Fourth Thread");
System.out.println("Parent Thread Group of tg1 is "+tg1.getParent());
System.out.println("Thread Group of t3 is "+t3.getThreadGroup());
tg1.setMaxPriority(7);
System.out.println("Thread Group of t1 is "+t1.getThreadGroup());
System.out.println("Thread Group of t3 is "+t3.getThreadGroup());
t1.start();
t2.start();
t3.start();
t4.start();
System.out.println("No: of Threads active in tg is "+tg.activeCount());
}
}
```

```
D:\Seshu\Thread>javac TGroups.java
D:\Seshu\Thread>java TGroups
Parent Thread Group of tg1 is java.lang.ThreadGroup[name=First Group,maxpri=10]
Thread Group of  t3 is java.lang.ThreadGroup[name=Second Group,maxpri=10]
Thread Group of  t1 is java.lang.ThreadGroup[name=First Group,maxpri=10]
Thread Group of  t3 is java.lang.ThreadGroup[name=Second Group,maxpri=7]
No: of Threads active in tg is 4
I am Cancellation Thread
I am Reservation Thread
I am Reservation Thread
I am Cancellation Thread
```

Name of the method	Function
ThreadGroup getThreadGroup()	Returns the reference of the ThreadGroup
public int getMaxPriority()	Returns maximum priority of the ThreadGroup
public void setMaxPriority(int priority)	Sets a new maximum priority for a ThreadGroup
String getName()	Returns as a String the name of the ThreadGroup
String getParent()	Determines the parent of a ThreadGroup

Input / Output

The io package supports Java's basic I/O (input/output) system, including file I/O. most real applications of Java are not text-based, console programs. Rather, they are graphically oriented programs that rely upon Java's Abstract Window Toolkit (AWT) or Swing for interaction with the user. Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not very important to Java programming. The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master.

Streams

Java programs perform I/O through streams. A stream is an abstraction that either producesor consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io package.

Byte Streams and Character Streams

Java defines two types of streams: byte and character. Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams. The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. This is why older code that doesn't use character streams should be updated to take advantage of them, where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters. An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: InputStream and OutputStream. Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. The byte stream classes are shown in Table 1. A few of these classes are discussed later in this section. Others are described in Part II. Remember, to use the stream classes, you must import java.io.

The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement. Two of the most important are read( ) and write( ), which, respectively, read and

write bytes of data. Both methods are declared as abstract inside InputStream and OutputStream. They are overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes, Reader and Writer. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes are shown in Table 2.

The abstract classes Reader and Writer define several key methods that the other stream classes implement. Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively. These methods are overridden by derived stream classes.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

TABLE 1 The Byte Stream Classes

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

TABLE 2 The Character Stream I/O Classes

The Predefined Streams

As you know, all Java programs automatically import the java.lang package. This package defines a class called System, which encapsulates several aspects of the run-time environment. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. System also contains three predefined stream variables: in, out, and err. These fields are declared as public, static, and final within System. This means that they can be used by any other part of your program and without reference to a specific System object.

System.out refers to the standard output stream. By default, this is the console. System.in refers to standard input, which is the keyboard by default. System.err refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream. These are byte streams, even though they typically are used to read and write characters from and to the console. As you will see, you can wrap these within character based streams, if desired.

The preceding chapters have been using System.out in their examples. You can use System.err in much the same way. As explained in the next section, use of System.in is a little more complicated.

Reading Console Input

In Java 1.0, the only way to perform console input was to use a byte stream, and older code that uses this approach persists. Today, using a byte stream to read console input is still technically possible, but doing

so is not recommended. The preferred method of reading console input is to use a character-oriented stream, which makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from System.in. To obtain a character based stream that is attached to the console, wrap System.in in a BufferedReader object.

BufferedReader supports a buffered input stream. Its most commonly used constructor is shown here:

BufferedReader(Reader inputReader)

Here, inputReader is the stream that is linked to the instance of BufferedReader that is being created. Reader is an abstract class. One of its concrete subclasses is InputStreamReader, which converts bytes to characters. To obtain an InputStreamReader object that is linked to System.in, use the following constructor:

InputStreamReader(InputStream inputStream)

Because System.in refers to an object of type InputStream, it can be used for inputStream. Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

After this statement executes, br is a character-based stream that is linked to the console through System.in.

Reading Characters

To read a character from a BufferedReader, use read( ). The version of read( ) that we will be using is

int read( ) throws IOException Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered. As you can see, it can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "q.” Notice that any I/O exceptions that might be generated are simply thrown out of main( ). Such an approach is common when reading from the console, but you can handle these types of errors yourself, if you chose.

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead {
     public static void main(String args[])throws IOException{
          char c;
          BufferedReader br = new
          BufferedReader(new InputStreamReader(System.in));
          System.out.println("Enter characters, 'q' to quit.");

          // read characters
          do {
               c = (char) br.read();
               System.out.println(c);
          } while(c != 'q');
     }
}
```

```
Here is a sample run:
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

This output may look a little different from what you expected, because System.in is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make read( ) particularly valuable for interactive console input.

Reading Strings

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

String readLine( ) throws IOException

As you can see, it returns a String object.

The following program demonstrates BufferedReader and the readLine( ) method; the program reads and displays lines of text until you enter the word "stop":

```
// Read a string from console using a BufferedReader.
import java.io.*;
class BRReadLines {
     public static void main(String args[])throws IOException{
          // create a BufferedReader using System.in
          BufferedReader br = new BufferedReader(new
          InputStreamReader(System.in));
          String str;
          System.out.println("Enter lines of text.");
          System.out.println("Enter 'stop' to quit.");
          do {
          str = br.readLine();
          System.out.println(str);
          } while(!str.equals("stop"));
     }
}
```

Writing Console Output

Console output is most easily accomplished with print( ) and println( ), described earlier, which are used in most of the examples in this book. These methods are defined by the class PrintStream (which is the type of object referenced by System.out). Even though System.out is a byte stream, using it for simple

program output is still acceptable. However, a character-based alternative is described in the next section.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is shown here:

void write(int byteval)

This method writes to the stream the byte specified by byteval. Although byteval is declared as an integer, only the low-order eight bits are written. Here is a short example that uses write( ) to output the character "A" followed by a newline to the screen:

```
// Demonstrate System.out.write().
class WriteDemo {
     public static void main(String args[]) {
          int b;
          b = 'A';
          System.out.write(b);
          System.out.write('\n');
     }
}
```

You will not often use write( ) to perform console output (although doing so might be useful in some situations), because print( ) and println( ) are substantially easier to use.

The PrintWriter Class

Although using System.out to write to the console is acceptable, its use is recommended mostly for debugging purposes or for sample programs, such as those found in this book. For real-world programs, the recommended method of writing to the console when using Java is through a PrintWriter stream. PrintWriter is one of the character-based classes.Using a character-based class for console output makes it easier to internationalize your program.

PrintWriter defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream outputStream, boolean flushOnNewline)

Here, outputStream is an object of type OutputStream, and flushOnNewline controls whether Java flushes the output stream every time a println( ) method is called. If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic.

PrintWriter supports the print( ) and println( ) methods for all types including Object. Thus, you can use these methods in the same way as they have been used with System.out. If an argument is not a simple type, the PrintWriter methods call the object's toString( ) method and then print the result.

To write to the console by using a PrintWriter, specify System.out for the output stream and flush the stream after each newline. For example, this line of code creates a PrintWriter that is connected to console output:

PrintWriter pw = new PrintWriter(System.out, true);

The following application illustrates using a PrintWriter to handle console output:

```
// Demonstrate PrintWriter
```

```
import java.io.*;
public class PrintWriterDemo {
     public static void main(String args[]) {
          PrintWriter pw = new PrintWriter(System.out, true);
          pw.println("This is a string");
          int i = -7;
          pw.println(i);
          double d = 4.5e-7;
          pw.println(d);
     }
}
The output from this program is shown here:
This is a string
-7
4.5E-7
```

Remember, there is nothing wrong with using System.out to write simple text output to the console when you are learning Java or debugging your programs. However, using a PrintWriter will make your real-world applications easier to internationalize. Because no advantage is gained by using a PrintWriter in the sample programs shown in this book, we will continue to use System.out to write to the console.

Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file. However, Java allows you to wrap a byte-oriented file stream within a character-based object.

Two of the most often-used stream classes are FileInputStream and FileOutputStream, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. While both classes support additional, overridden constructors, the following are the forms that we willbe using:

FileInputStream(String fileName) throws FileNotFoundException

FileOutputStream(String fileName) throws FileNotFoundException

Here, fileName specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then FileNotFoundException is thrown. For output streams, if the file cannot be created, then FileNotFoundException is thrown. When an output file is opened, any preexisting file by the same name is destroyed.

When you are done with a file, you should close it by calling close( ). It is defined by both FileInputStream and FileOutputStream, as shown here:

void close( ) throws IOException

To read from a file, you can use a version of read( ) that is defined within FileInputStream. The one that we will use is shown here:

int read( ) throws IOException

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. read( ) returns –1 when the end of the file is encountered. It can throw an IOException.

The following program uses read( ) to input and display the contents of a text file, the name of which is specified as a command-line argument. Note the try/catch blocks that handle two errors that might occur when this program is used—the specified file not being found
or the user forgetting to include the name of the file. You can use this same approach whenever you use command-line arguments. Other I/O exceptions that might occur are simply thrown out of main( ), which is acceptable for this simple example. However, often you will want to handle all I/O exceptions yourself when working with files.

```
/* Display a text file.
To use this program, specify the name of the file that you want to see.
For example, to see a file called TEST.TXT, use the following command line.
java ShowFile TEST.TXT
*/
import java.io.*;
class ShowFile {
     public static void main(String args[]) throws IOException{
          int i;
          FileInputStream fin;
          try {
               fin = new FileInputStream(args[0]);
          } catch(FileNotFoundException e) {
               System.out.println("File Not Found");
               return;
          } catch(ArrayIndexOutOfBoundsException e) {
               System.out.println("Usage: ShowFile File");
               return;
          }

          // read characters until EOF is encountered
          do {
               i = fin.read();
               if(i != -1) System.out.print((char) i);
          } while(i != -1);
          fin.close();
     }
}
```

To write to a file, you can use the write( ) method defined by FileOutputStream. Its simplest form is shown here:

void write(int byteval) throws IOException

This method writes the byte specified by byteval to the file. Although byteval is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an IOException is thrown. The next example uses write( ) to copy a text file:

```
/* Copy a text file.
To use this program, specify the name of the source file and the
destination file. For example, to copy a file called FIRST.TXT to a
file called SECOND.TXT, use the following command line.
java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;
class CopyFile {
     public static void main(String args[])throws IOException{
          int i;
          FileInputStream fin;
          FileOutputStream fout;
          try {
          // open input file
          try {
               fin = new FileInputStream(args[0]);
          } catch(FileNotFoundException e) {
               System.out.println("Input File Not Found");
               return;
          }
          // open output file
          try {
               fout = new FileOutputStream(args[1]);
          } catch(FileNotFoundException e) {
               System.out.println("Error Opening Output File");
               return;
          }
          } catch(ArrayIndexOutOfBoundsException e) {
               System.out.println("Usage: CopyFile From To");
               return;
          }
          // Copy File
          try {
               do {
                    i = fin.read();
                    if(i != -1) fout.write(i);
               } while(i != -1);
               } catch(IOException e) {
               System.out.println("File Error");
          }
          fin.close();
```

```
            fout.close();
        }
}
```

Notice the way that potential I/O errors are handled in this program. Unlike some other computer languages, including C and C++, which use error codes to report file errors, Java uses its exception handling mechanism. Not only does this make file handling cleaner, but it also enables Java to easily differentiate the end-of-file condition from file errors when input is being performed. In C/C++, many input functions return the same value when an error occurs and when the end of the file is reached. (That is, in C/C++, an EOF condition often is mapped to the same value as an input error.) This usually means that the programmer must include extra program statements to determine which event actually occurred. In Java, errors are passed to your program via exceptions, not by values returned by read( ). Thus, when read( ) returns –1, it means only one thing: the end of the file has been encountered.