UNIT 1 MULTITHREADED PROGRAMMING

1.0 INTRODUCTION

A thread is single sequence of execution that can run independently in an application. This unit covers the very important concept of multithreading in programming. Uses of thread in programs are good in terms of resource utilization of the system on which application(s) is running. Multithreaded programming is very useful in network and Internet applications development. In this unit you will learn what is multithreading, how thread works, how to write programs in Java using multithreading. Also, in this unit will be explained about thread-properties, synchronization, and interthread communication.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- describe the concept of multithreading;
- explain the Java thread model;
- create and use threads in program;
- describe how to set the thread priorities;
- use the concept of synchronization in programming, and
- use inter-thread communication in programs.

1.2 MULTITHREADING: AN INTRODUCTION

Multithreaded programs support more than one concurrent thread of execution. This means they are able to simultaneously execute multiple sequences of instructions. Each instruction sequence has its own unique flow of control that is independent of all others. These independently executed instruction sequences are known as *threads*. Your PC has only a single CPU; you might ask how it can execute more than one thread at the same time? In single processor systems, only a single thread of execution occurs at a given instant. But multiple threads in a program increase the utilization of CPU.

The CPU quickly switches back and forth between several threads to create an illusion that the threads are executing at the same time. You know that single-processor systems support logical concurrency only. Physical concurrency is not supported by it. Logical concurrency is the characteristic exhibited when multiple threads execute with separate, independent flow of control. On the other hand on a multiprocessor system, several threads can execute at the same time, and physical concurrency is achieved.

The advantage of multithreaded programs is that they support logical concurrency. Many programming languages support multiprogramming, as it is the logically concurrent execution of multiple programs. For example, a program can request the operating system to execute programs A, B and C by having it spawn a separate process for each program. These programs can run in a concurrent manner, depending upon the multiprogramming features supported by the underlying operating system. Multithreading differs from multiprogramming. Multithreading provides concurrency within the content of a single process. But multiprogramming also provides concurrency between processes. Threads are not complete processes in themselves. They are a separate flow of control that occurs within a process.

A process is the operating system object that is created when a program is executed

In *Figure1a and Figure 1 b* the difference between multithreading and multiprogramming is shown.

Multiprogramming:

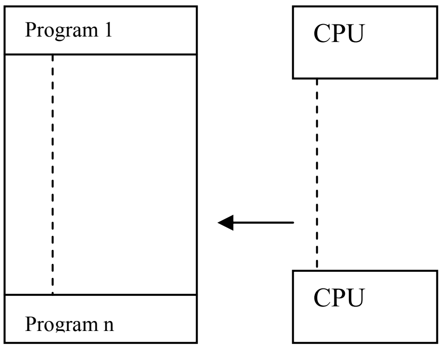


Figure 1a: Multiprogramming

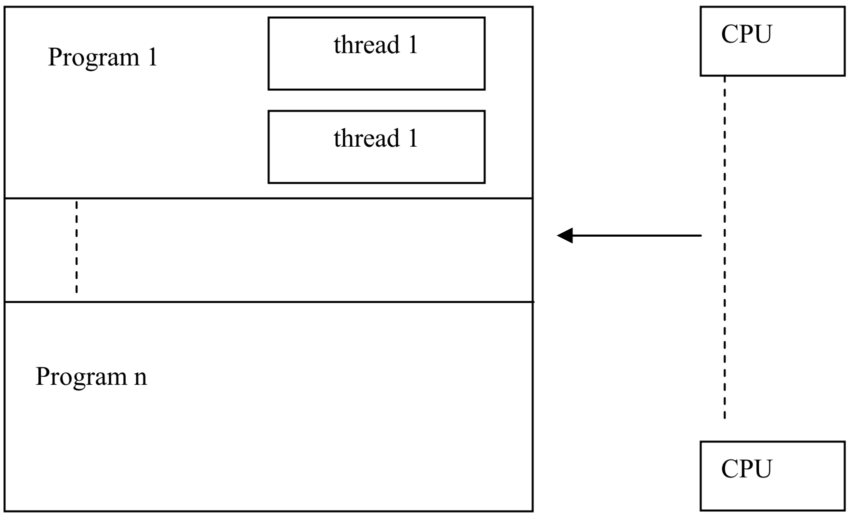


Figure 1b: Multithreading

Now let us see the advantages of multithreading.

Advantages of Multithreading

The advantages of multithreading are:

i. Concurrency can be used within a process to implement multiple instances of simultaneous services.
ii. Multithreading requires less processing overhead than multiprogramming because concurrent threads are able to share common resources more efficiently.

A multithreaded web server is one of the examples of multithreaded programming. Multithreaded web servers are able to efficiently handle multiple browser requests. They handle one request per processing thread.

iii. Multithreading enables programmers to write very efficient programs that make maximum use of the CPU. Unlike most other programming languages, Java provides built-in support for multithreaded programming. The Java run-time system depends on threads for many things. Java uses threads to enable the entire environment to be synchronous.

Now let us see how Java provides multithreading.

1.3 THE MAIN THREAD

When you start any Java program, one thread begins running immediately, which is called the ***main thread*** of that program. Within the main thread of any Java program you can create other 'child' threads. Every thread has its lifetime. During programming you should take care that the main thread is the last thread to finish execution because it performs various shutdown actions. In fact for some older JVM's if the main thread finishes before a child thread, then the Java run-time system may *hang*.

The main thread is created automatically when your program is started. The main thread of Java programs is controlled through an object of ***Thread*** *class*.

You can obtain a reference to Thread class object by calling the method current Thread( ), which is a static method:
static Thread currentThread( )

By using this method, you get a reference to the thread in which this method is called. Once you have a reference to the main thread, you can control it just like other threads created by you. In coming sections of this unit you will learn how to control threads.

Now let us see the program given below to obtain a reference to the main thread and the name of the main thread is set to MyFirstThread using setName (String) method of *Thread* class

```
//program
class Thread_Test
{
public static void main (String args [ ])
{
try
{
Thread threadRef  = Thread.currentThread( );
System.out.println("Current Thread :"+ threadRef);
System.out.println ("Before changing the name of main thread : "+ threadRef);
//change the internal name of the thread
threadRef.setName("MyFirstThread");
System.out.println ("After changing the name of main thread : "+threadRef);
}
catch (Exception e)
{
System.out.println("Main thread interrupted");
  }
}
}
```

Output:

```
Current Thread: Thread[main,5,main]
Before changing the name of main thread: Thread[main,5,main]
After changing the name of main thread: Thread[MyFirstThread,5,main]
```

In output you can see in square bracket ([]) the name of the thread, thread priority which is 5, by default and main is also 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. The particular run-time environment manages this process. You can see that the name of the thread is changed to MyFirstThread and is displayed in output.

☞ Check Your Progress 1

1) How does multithreading take place on a computer with a single CPU?

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………

2) State the advantages of multithreading.

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………

3) How will you get reference to the main thread?

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………

4) What will happen if the main thread finishes before the child thread?

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………

The Java run-time system depends on threads for many activities, and all the class libraries are designed with multithreading in mind. Now let us see the Java thread model.

1.4 JAVA THREAD MODEL

The benefit of Java's multithreading is that a thread can pause without stopping other parts of your program. A paused thread can restart. A thread will be referred to as dead when the processing of the thread is completed. After a thread reaches the dead state, then it is not possible to restart it.

The thread exists as an object; threads have several well-defined states in addition to the dead states. These state are:

Ready State

When a thread is created, it doesn't begin executing immediately. You must first invoke start ( ) method to start the execution of the thread. In this process the thread scheduler must allocate CPU time to the Thread. A thread may also enter the ready state if it was stopped for a while and is ready to resume its execution.

Running State

Threads are born to run, and a thread is said to be in the running state when it is actually executing. It may leave this state for a number of reasons, which we will discuss in the next section of this unit.

Waiting State

A running thread may perform a number of actions that will cause it to wait. A common example is when the thread performs some type of input or output operations.

As you can see in *Figure 2* given below, a thread in the waiting state can go to the ready state and from there to the running state. Every thread after performing its operations has to go to the dead state.

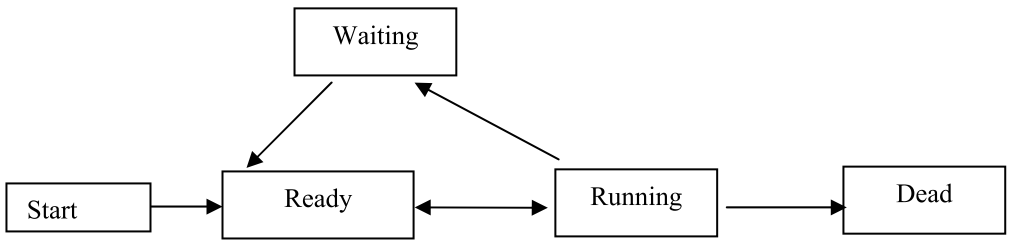


Figure 2: The Thread States

A thread begins as a ready thread and then enters the running state when the thread scheduler schedules it. The thread may be prompted by other threads and returned to the ready state, or it may wait on a resource, or simply stop for some time. When this happens, the thread enters the waiting state. To run again, the thread must enter the ready state. Finally, the thread will cease its execution and enter the dead state.

The multithreading system in Java is built upon the **Thread Class**, its methods and its companion interface, **Runnable**. To create a new thread, your program will either *extend Thread Class* or *implement the Runnable interface*. The Thread Class defines several methods that help in managing threads.

For example, if you have to create your own thread then you have to do one of the following things:

1)

```
class MyThread extends Thread
 {
  MyThread(arguments) // constructor
   {
   }//initialization
   public void run()
   {
   // perform operations
   }
   }
```

Write the following code to create a thread and start it running:

```
     MyThread p = new MyThread(arguments);
     p.start();
```

2)

```
    class MyThread implements Runnable
     {
    MyThread(arguments)
    {
     //initialization
    }
   public void run()
   {
  // perform operation
   }
   }
```

The Thread Class Constructors

The following are the Thread class constructors:

Thread()
Thread(Runnable target)
Thread (Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Thread(ThreadGroup group, String name)

Thread Class Method

Some commonly used methods of Thread class are given below:

static Thread **currentThread**() Returns a reference to the currently executing thread object.

String **getName**() Returns the name of the thread in which it is called

int **getPriority**() Returns the Thread's priority

void **interrupt**() Used for Interrupting the thread.

static boolean **interrupted**() Used to tests whether the current thread has been interrupted or not.

boolean **isAlive**() Used for testing whether a tread is alive or not.

boolean **isDaemon**() Used for testing whether a thread is a daemon thread or not.

void **setName**(String NewName) Changes the name of the thread to NewName

void **setPriority**(int newPriority) Changes the priority of thread.

static void **sleep**(long millisec) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

void **start**() Used to begin execution of a thread .The Java Virtual Machine calls the run method of the thread in which this method is called.

String **toString**() Returns a string representation of thread.String includes the thread's name, priority, and thread group.

static void **yield**() Used to pause temporarily to currently executing thread object and allow other threads to execute.

static int **activeCount**() Returns the number of active threads in the current thread's thread group.

void **destroy**() Destroys the thread without any cleanup.

Creating Threads

Java provides native support for multithreading. This support is centered on the Java.lang.Thread class, the Java.lang.Runnable interface and methods of the Java.lang.object class. In Java, support for multithreaded programming is also provided through synchronized methods and statements.

The thread class provides the capability to create thread objects, each with its own separate flow of control. The thread class encapsulates the data and methods associated with separate threads of execution and enables multithreading to be integrated within Java's object oriented framework. The minimal multithreading support required of the Thread Class is specified by the java.lang.Runnable interface. This interface defines a single but important method run.
public void run( )

This method provides the entry point for a separate thread of execution. As we have discussed already Java provides two approaches for creating threads. In the first approach, you create a subclass of the Thread class and override the run( ) method to provide an entry point for the Thread's execution. When you create an instance of subclass of Thread class, you invoke start( ) method to cause the thread to execute as an independent sequence of instructions. The start( ) method is inherited from the Thread class. It initializes the thread using the operating system's multithreading capabilities, and invokes the run( ) method.

Now let us see the program given below for creating threads by inheriting the Thread class.

```
//program
class MyThreadDemo extends Thread
{
public String MyMessage [ ]=
{"Java","is","very","good","Programming","Language"};
MyThreadDemo(String s)
{
```

```
super(s);
}
public void run( )
{
String name = getName( );
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
 sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo
{
public static void main ( String args [ ])
{
MyThreadDemo            Td1= new MyThreadDemo("thread 1:");
MyThreadDemo            Td2= new MyThreadDemo("thread 2:");
Td1.start ( );
Td2.start ( );
boolean isAlive1 =  true;
boolean  isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
 isAlive1= false;
 System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
 isAlive2= false;
 System.out.println ("Thread 2 is dead");
}
 }
 while(isAlive1 || isAlive2);
}
}
```

Output:

```
thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
thread 2::Language
Thread 1 is dead
Thread 2 is dead
```

This output shows how two threads execute in sequence, displaying information on the console. The program creates two threads of execution, Td 1 and Td2. The threads display the "Java","is","very","good","Programming","Language" message word by word, while waiting a short amount of time between each word. Because both threads share the console window, the program's output identifies which thread wrote to the console during the program's execution.

Now let us see how threads are created in Java by implementing the java.lang.Runnable interface. The Runnable interface consists of only one method, i.e, run ( )method that provides an entry point into your threads execution.

In order to run an object of class you have created, as an independent thread, you have to pass it as an argument to a constructor of class Thread.

```
//program
class MyThreadDemo  implements  Runnable
{
public String MyMessage [ ]=
{"Java","is","very","good","Programming","Language"};
String name;
public MyThreadDemo(String s)
{
name = s;
}
public void run( )
{
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
 Thread.currentThread().sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo1
{
public static void main ( String args [ ])
{
Thread  Td1= new Thread( new MyThreadDemo("thread 1:"));
Thread  Td2= new Thread(new MyThreadDemo("thread 2:"));
Td1.start ( );
Td2.start ( );
boolean isAlive1 =  true;
boolean  isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
 isAlive1= false;
 System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
 isAlive2= false;
 System.out.println ("Thread 2 is dead");
}
 }
 while(isAlive1 || isAlive2);
}
}
```

Output:

```
thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
Thread 1 is dead
thread 2::Language
Thread 2 is dead
```

This program is similar to previous program and also gives same output. The advantage of using the Runnable interface is that your class does not need to extend the thread class. This is a very helpful feature when you create multithreaded applets. The only disadvantage of this approach is that you have to do some more work to create and execute your own threads.

☞ Check Your Progress 2

		T	F
1)	Sate True/False for the following statements:		
	i. Initial state of a newly created thread is Ready state.	☐	
	ii. Ready, Running, Waiting and Dead states are thread states.	☐	
	iii. When a thread is in execution it is in Waiting state.	☐	
	iv. A dead thread can be restarted.	☐	
	v. sart() method causes an object to begin executing as a separate thread.	☐	
	vi. run( ) method must be implemented by all threads.	☐	

2) Explain how a thread is created by extending the Thread class.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) Explain how threads are created by implementing Runnable interface

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

Java provides methods to assign different priorities to different threads. Now let us discuss how thread priority is handled in Java.

1.5 THREAD PRIORITIES

Java assigns to each *thread* a *priority***.** Thread priority determines how a thread should be treated with respect to others. Thread priority is an integer that specifies the relative priority of one thread to another. A thread's priority is used to decide which thread. A thread can voluntarily relinquish control. Threads relinquish control by explicitly yielding, sleeping, or blocking on pending Input/ Output operations. In this scenario, all other threads are examined, and the highest- priority thread that is ready to run gets the chance to use the CPU.

A higher-priority thread can pre-empt a low priority thread. In this case, a lower-priority thread that does not yield the processor is forcibly pre-empted. In cases where two threads with the same priority are competing for CPU cycles, the situation is handled differently by different operating systems. In Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. In case of Solaris, threads of equal priority must voluntarily yield control to their peers. If they don't the other threads will not run.

Java thread class has defined two constants MIN_PRIORITY and MAX_PRIORITY. Any thread priority lies between MIN_PRIORITY and MAX_PRIORITY.Currently, the value of MIN_PRIORITY is 1 and MAX_PRIORITY is 10.

The priority of a thread is set at the time of creation. It is set to the same priority as the Thread that created it. The default priority of a thread can be changed by using the setPriority( ) method of Thread class .

Final void setPriority (int Priority_Level) where Priority_Level specifies the new priority for the calling thread. The value of Priority_Level must be within the range MIN_PRIORITY and MAX_PRIORITY.

To return a thread to default priority, specify Norm_Priority, which is currently 5. You can obtain the current priority setting by calling the getPriority( ) method of thread class.
final int getPriority( ).

Most operating systems support one of two common approaches to thread scheduling.

Pre-emptive Scheduling: The highest priority thread continues to execute unless it dies, waits, or is pre-empted by a higher priority thread coming into existence.

Time Slicing: A thread executes for a specific slice of time and then enters the ready state. At this point, the thread scheduler determines whether it should return the thread to the ready state or schedule a different thread.

Both of these approaches have their advantages and disadvantages. Pre-emptive scheduling is more predictable. However, its disadvantage is that a higher priority thread could execute forever, preventing lower priority threads from executing. The time slicing approach is less predictable but is better in handling selfish threads. Windows and Macintosh implementations of Java follow a time slicing approach. Solaris and other Unix implementations follow a pre-emptive scheduling approach.

Now let us see this example program, it will help you to understand how to assign thread priority.

```
//program
class Thread_Priority
{
public static void main (String args [ ])
{
 try
{
Thread Td1  = new  Thread("Thread1");
Thread Td2  = new  Thread("Thread2");
System.out.println ("Before any change in default priority:");
System.out.println("The Priority of "+Td1.getName() +" is "+ Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority());
//change in priority
Td1.setPriority(7);
Td2.setPriority(8);
System.out.println ("After changing in Priority:");
System.out.println("The Priority of "+Td1.getName() +" is "+ Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority());
}
catch ( Exception e)
{
  System.out.println("Main thread interrupted");
 }
}
}
```

Output:

```
Before any change in default priority:
The Priority of Thread1 is 5
The Priority of Thread1 is 5
After changing in Priority:
The Priority of Thread1 is 7
The Priority of Thread1 is 8
```

Multithreaded programming introduces an asynchronous behaviour in programs. Therefore, it is necessary to have a way to ensure synchronization when program need it. Now let us see how Java provide synchronization.

1.6 SYNCHRONIZATION IN JAVA

If you want two threads to communicate and share a complicated data structure, such as a linked list or graph, you need some way to ensure that they don't conflict with each other. In other words, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements model of ***interprocess synchronizations.*** You know that once a thread enters a monitor, all other threads must wait until that thread exists in the monitor. Synchronization support is built in to the Java language.

There are many situations in which multiple threads must share access to common objects. There are times when you might want to coordinate access to shared resources. For example, in a database system, you would not want one thread to be updating a database record while another thread is trying to read from the database. Java enables you to coordinate the actions of multiple threads using synchronized methods and synchronized statements. Synchronization provides a simple monitor facility that can be used to provide mutual-exclusion between Java threads.

The monitor is a control mechanism

Once you have divided your program into separate threads, you need to define how they will communicate with each other. Synchronized methods are used to coordinate access to objects that are shared among multiple threads. These methods are declared with the *synchronized* keyword. Only one synchronized method at a time can be invoked for an object at a given point of time. When a synchronized method is invoked for a given object, it acquires the monitor for that object.In this case no other synchronized method may be invoked for that object until the monitor is released. This keeps synchronized methods in multiple threads without any conflict with each other.

When a synchronized method is invoked for a given object, it tries to acquire the lock for that object. If it succeeds, no other synchronized method may be invoked for that object until the lock is released. A lock is automatically released when the method completes its execution and returns. A lock may also be released when a synchronized method executes certain methods, such as wait( ).

Now let us see this example which explains how synchronized methods and object locks are used to coordinate access to a common object by multiple threads.

```
//program

class Call_Test
{
synchronized void callme (String msg)
{
//This prevents other threads from entering call( ) while another thread is using it.
System.out.print ("["+msg);
try
{
Thread.sleep (2000);
}
catch ( InterruptedException e)
{
System.out.println ("Thread is Interrupted");
}
System.out.println ("]");
}
}

class Call implements Runnable
```

```
{
String msg;
Call_Test ob1;
Thread t;
public Call (Call_Test tar, String s)
{
System.out.println("Inside caller method");
ob1= tar;
msg = s;
t = new Thread(this);
t.start();
}
public void run( )
{
ob1.callme(msg);
}
}

class Synchro_Test
{
public static void main (String args [ ])
{
Call_Test T= new Call_Test( );
Call ob1= new Call (T, "Hi");
Call ob2= new Call (T, "This ");
Call ob3= new Call (T, "is");
Call ob4= new Call (T, "Synchronization");
Call ob5= new Call (T, "Testing");
try
{
ob1.t.join( );
ob2.t.join( );
ob3.t.join( );
ob4.t.join( );
ob5.t.join( );
}
catch ( InterruptedException e)
{
System.out.println (" Interrupted");
}
}
}
```

Output:

```
Inside caller method
Inside caller method
Inside caller method
Inside caller method
Inside caller method
[Hi]
[This ]
[is]
[Synchronization]
[Testing]
```

If you run this program after removing synchronized keyword, you will find some output similar to:

```
Inside caller method
Inside caller method
Inside caller method
Inside caller method
[Hi[This [isInside caller method
[Synchronization[Testing]
]
]
]
]
```

This result is because when in program sleep( ) is called , the callme( ) method allows execution to switch to another thread. Due to this, output is a mix of the three message strings.

So if at any time you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should make these methods synchronized. It helps in avoiding conflict.

An effective means of achieving synchronization is to create synchronized methods within classes. But it will not work in all cases, for example, if you want to synchronize access to object's of a class that was not designed for multithreaded programming or the class does not use synchronized methods. In this situation, the *synchronized statement* is a solution. Synchronized statements are similar to synchronized methods. It is used to acquire a lock on an object before performing an action.

The synchronized statement is different from a synchronized method in the sense that it can be used with the lock of any object and the synchronized method can only be used with its object's (or class's) lock. It is also different in the sense that it applies to a statement block, rather than an entire method. The syntax of the synchronized statement is as follows:

```
synchronized (object)
{
// statement(s)
}
```

The statement(s) enclosed by the braces are only executed when the current thread acquires the lock for the object or class enclosed by parentheses.

Now let us see how interthread communication takes place in java.

1.7 INTERTHREAD COMMUNICATION

To avoid wastage of precious time of CPU, or to avoid polling, Java includes an interprocess communication mechanism via the *wait( ), notify( ),* and *notifyAll( )* methods. These methods are implemented as final methods, so all classes have them. These three methods can be called only from within a synchronized method.

wait( ): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll().
notify( ): Wakes up a single thread that is waiting on this object's monitor.
notifyAll( ): Wakes up all threads that are waiting on this object's monitor, the highest priority Thread will be run first.

These methods are declared within objects as following:
final void wait( ) throws Interrupted Exception

```
final void notify( )
final void notifyAll( )
```

These methods enable you to place threads in a waiting pool until resources become available to satisfy the Thread's request. A separate resource monitor or controller process can be used to notify waiting threads that they are able to execute.

Let us see this example program written to control access of resource using wait() and notify () methods.

```
//program
class WaitNotifyTest implements Runnable
{
WaitNotifyTest ()
{
Thread th = new Thread (this);
th.start();
}
synchronized void notifyThat ()
{
System.out.println ("Notify the threads waiting");
this.notify();
}
synchronized public void run()
{
try
{
System.out.println("Thead is waiting....");
this.wait ();
}
catch (InterruptedException e){}
System.out.println ("Waiting thread notified");
}
}
class runWaightNotify
{
public static void main (String args[])
{
WaitNotifyTest wait_not = new WaitNotifyTest();
Thread.yield ();
wait_not.notifyThat();
}
}
```

Output:

```
Thead is waiting....
Notify the threads waiting
Waiting thread notified
```

☞ Check Your Progress 3

1) Explain the need of synchronized method

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) Run the program given below and write output of it:

```
//program
class One
{
public static void main (String arys[])
{
Two t = new Two( );
t. Display("Thread is created.");
t.start();
}
}
class Two extends Thread
{
public void Display(String s)
{
System.out.println(s);
}
public void run ()
{
System.out.println("Thread is running");
}
}
```

3) What is the output of the following program:

```
//program
class ThreadTest1
{
public static void main(String arrays [ ])
{
Thread1 th1 = new Thread1("q1");
Thread1 th2 = new Thread1("q2");
th1.start ();
th2.start ();
}
}
class Thread1 extends Thread
{
public void run ()
{
for (int i = 0; i <3 ;++i)
{
try
{
System.out.println("Thread: "+this.getName()+" is sleeping");
this.sleep (2000);
}
catch (InterruptedException e)
{
System.out.println("interrupted");
}
}
}
public Thread1(String s)
{
super (s);
}
}
```

4) Run this program and explain the output obtained

```
//program
public class WaitNotif
{
int i=0;
public static void main(String argv[])
{
WaitNotif ob = new WaitNotif();
ob.testmethod();
}
public void testmethod()
{
while(true)
{
try
{
wait();
}
catch (InterruptedException e)
{
System.out.println("Interrupted Exception");
}
}
}//End of testmethod
}
```

1.8 SUMMARY

This unit described the working of multithreading in Java. Also you have learned what is the main thread and when it is created in a Java program. Different states of threads are described in this unit. This unit explained how threads are created using Thread class and Runnable interface. It explained how thread priority is used to determine which thread is to execute next. This unit explains concept of synchronization, creating synchronous methods and inter thread communication. It is also explained how object locks are used to control access to shared resources.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) In single CPU system, the task scheduler of operating system allocates executions time to multiple tasks. By quickly switching between executing tasks, it creates the impression that tasks executes concurrently.

2) i Provide concurrent execution of multiple instances of different services.
   ii Better utilization of CPU time.

3) To get reference of main thread in any program write:
   Thread mainthreadref = Thread.currentThread();
   As first executable statement of the program.

4) There is a chance that Java runtime may hang.

Check Your Progress 2

1) i True
   ii True

iii False
iv False
v True
vi True

2) A thread can be constructed on any object that implements Runnable interface. If you have to create a thread by implementing Runnable interface, implement the only method run() of this interface. This can be done as:

```
class  MyThread implements Runnable
{
 //Body
public void run()// this is method of Runnable interface
{
//method body
}
}
```

3) One way to create a thread is, to create a new class that extends Thread class. The extended class must override the run method of Thread class. This run() method work as entry point for the newly created thread. Also in extended class must be start method to start the execution. This can be done as:

```
class  My Thread extends Thread
{
MyThread()
{
super("Name of Thread");
System.out.println("This is a new Thread");
Start();
}
public void run() // override Thread class run() method
{
//body
}
}
```

Check Your Progress 3

1) If there is a task to be performed by only one party at a time then some control mechanism is required to ensure that only one party does that activity at a time. For example if there is a need to access a shared resource with ensuring that resource will be used by only one thread at a time, then the method written to perform this operation should be declared as synchronized. Making such methods synchronized work in Java for such problem because if a thread is inside a synchronized method al other thread looking for this method have to wait until thread currently having control over the method leaves it.

2) Output:

```
Thread is created.
Thread is running
```

3) Output:

```
Thread: q1 is sleeping
Thread: q2 is sleeping
Thread: q1 is sleeping
Thread: q2 is sleeping
```

```
Thread: q1 is sleeping
Thread: q2 is sleeping
```

4) Output:

```
java.lang.IllegalMonitorStateException
at java.lang.Object.wait(Native Method)
at java.lang.Object.wait(Object.java:420)
at WaitNotif.testmethod(WaitNotif.java:16)
at WaitNotif.main(WaitNotif.java:8)
```

Exception in thread "main"

This exception occurs because of wait and notify methods are not called within synchronized methods.

UNIT 2 I/O IN JAVA

Structure **Page Nos.**

2.0 INTRODUCTION

Input is any information that is needed by a program to complete its execution. *Output* is any information that the program must convey to the user. Input and Output are essential for applications development.

To accept input a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. Whether reading data from a file or from a socket, the concept of serially reading from, and writing to, different data sources is the same. For that very reason, it is essential to understand the features of top-level classes (Java.io.Reader, Java.io.Writer).

In this unit you will be working on some basics of I/O (Input–Output) in Java such as Files creation through *streams* in Java code. A stream is a linear, sequential flow of bytes of input or output data. Streams are written to the file system to create files. Streams can also be transferred over the Internet.

In this unit you will learn the basics of Java streams by reviewing the differences between byte and character streams, and the various stream classes available in the Java.io package. We will cover the standard process for standard Input (Reading from console) and standard output (writing to console).

2.1 OBJECTIVES

After going through this unit you will be able to:

- explain basics of I/O operations in Java;
- use stream classes in programming;
- take inputs from console;
- write output on console;
- read from files, and
- write to files.

2.2 I/O BASICS

Java input and output are based on the use of *streams*, or sequences of bytes that travel from a source to a destination over a communication path. If a program is writing to a stream, you can consider it as a stream's *source*. If it is reading from a stream, it is the stream's *destination*. The communication path is dependent on the type of I/O being performed. It can consist of memory-to-memory transfers, a file system, a network, and other forms of I/O.

Streams are powerful because they abstract away the details of the communication path from input and output operations. This allows all I/O to be performed using a common set of methods. These methods can be extended to provide higher-level custom I/O capabilities.

Three streams given below are created automatically:

- System.out - standard output stream
- System.in - standard input stream
- System.err - standard error

An **InputStream** represents a stream of data from which data can be read. Again, this stream will be either directly connected to a device or else to another stream.

An **OutputStream** represents a stream to which data can be written. Typically, this stream will either be directly connected to a device, such as a file or a network connection, or to another output stream.

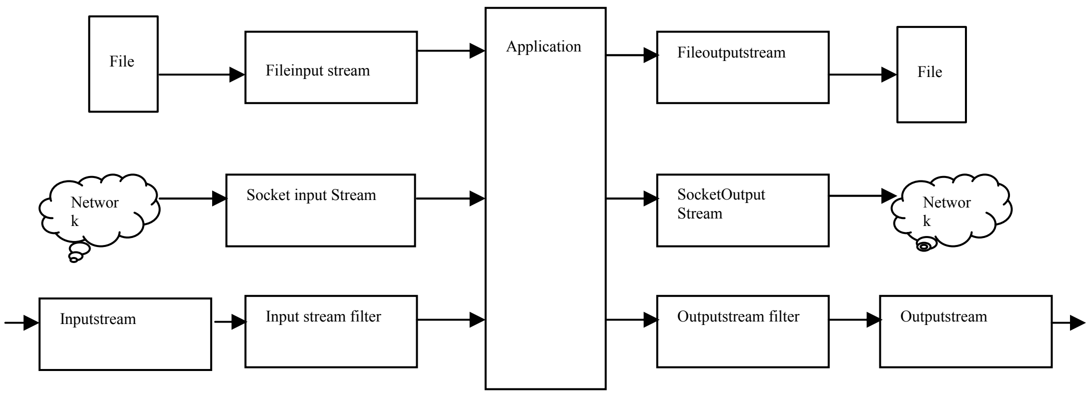


Figure 1: I/O stream basics

Java.io package

This package provides support for basic I/O operations. When you are dealing with the Java.io package some questions given below need to be addressed.

- What is the file format: text or binary?
- Do you want random access capability?
- Are you dealing with objects or non-objects?
- What are your sources and sinks for data?
- Do you need to use filtering (You will know about it in later section of this unit)?

For example:

- If you are using binary data, such as integers or doubles, then use the InputStream and OutputStream classes.

- If you are using text data, then the Reader and Writer classes are right.

Exceptions Handling during I/O

Almost every input or output method throws an exception. Therefore, any time you do an I/O operation, the program needs to catch exceptions. There is a large hierarchy of I/O exceptions derived from IOException class. Typically you can just catch IOException, which catches all the derived class exceptions. However, some exceptions thrown by I/O methods are not in the IOException hierarchy, so you should be careful about exception handling during I/O operations.

2.3 STREAMS AND STREAM CLASSES

The Java model for I/O is entirely based on streams.

There are two types of streams: byte streams and character streams.

Byte streams carry integers with values that range from 0 to 255. A diversified data can be expressed in byte format, including numerical data, executable programs, and byte codes – the class file that runs a Java program.

Character Streams are specialized type of byte streams that can handle only textual data.

Most of the functionality available for byte streams is also provided for character streams. The methods for character streams generally accept parameters of data type *char,* while *byte* streams work with *byte* data types. The names of the methods in both sets of classes are almost identical except for the suffix, that is, character-stream classes end with the suffix Reader or Writer and byte-stream classes end with the suffix InputStream and OutputStream.

For example, to read files using character streams use the Java.io.FileReader class, and for reading it using byte streams use Java.io.FileInputStream.

Unless you are writing programs to work with binary data, such as image and sound files, use readers and writers (character streams) to read and write information for the following reasons:

- They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
- They are easier to internationalize because they are not dependent upon a specific character encoding.
- They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

Now let us discuss byte stream classes and character stream classes one by one.

2.3.1 Byte Stream Classes

Java defines two major classes of byte streams: InputStream and OutputStream. To provide a variety of I/O capabilities subclasses are derived from these InputStream and OutputStream classes.

InputStream class

The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when created. The close() method can explicitly close a stream.

Methods of InputStream class

The basic method for getting data from any InputStream object is the read()method. public abstract int read() throws IOException: reads a single byte from the input stream and returns it.

public int read(byte[] bytes) throws IOException: fills an array with bytes read from the stream and returns the number of bytes read.

public int read(byte[] bytes, int offset, int length) throws IOException: fills an array from stream starting at position offset, up to length bytes. It returns either the number of bytes read or -1 for end of file.

public int available() throws IOException: the readmethod always blocks when there is no data available. To avoid blocking, program might need to ask ahead of time exactly how many bytes can safely read without blocking. The available method returns this number.

public long skip(long n): the skip() method skips over n bytes (passed as argument of skip()method) in a stream.

public synchronized void mark (int readLimit): this method marks the current position in the stream so it can backed up later.

OutputStream class

The OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when created. An Output stream can be explicitly closed with the close() method.

Methods of OutputStream class

public abstract void write(int b) throws IOException: writes a single byte of data to an output stream.

public void write(byte[] bytes) throws IOException: writes the entire contents of the bytes array to the output stream.

public void write(byte[] bytes, int offset, int length) throws IOException: writes length number of bytes starting at position offset from the bytes array.

The Java.io package contains several subclasses of InputStream and OutputStream that implement specific input or output functions. Some of these classes are:

- FileInputStream and FileOutputStream: Read data from or write data to a file on the native file system.
- PipedInputStream and PipedOutputStream : Implement the input and output components of a pipe. Pipes are used to channel the output from one program (or thread) into the input of another. A PipedInputStream must be connected to a PipedOutputStream and a PipedOutputStream must be connected to a PipedInputStream.
- ByteArrayInputStream and ByteArrayOutputStream : Read data from or write data to a byte array in memory.

ByteArrayOutputStream provides some additional methods not declared for OutputStream. The reset() method resets the output buffer to allow writing to restart at the beginning of the buffer. The size() method returns the number of bytes that have been written to the buffer. The write to () method is new.

- SequenceInputStream: Concatenate multiple input streams into one input stream.

- StringBufferInputStream: Allow programs to read from a StringBuffer as if it were an input stream.

Now let us see how Input and Output is being handled in the program given below: this program creates a file and writes a string in it, and reads the number of bytes in file.

```
// program for I/O
import Java.lang.System;
import Java.io.FileInputStream;
import Java.io.FileOutputStream;
import Java.io.File;
import Java.io.IOException;
public class FileIOOperations {
public static void main(String args[]) throws IOException {
 // Create output file test.txt
FileOutputStream outStream = new FileOutputStream("test.txt");
String s = "This program is for Testing I/O Operations";
for(int i=0;i<s.length();++i)
outStream.write(s.charAt(i));
outStream.close();
// Open test.txt for input
FileInputStream inStream = new FileInputStream("test.txt");
int inBytes = inStream.available();
System.out.println("test.txt has "+inBytes+" available bytes");
byte inBuf[] = new byte[inBytes];
int bytesRead = inStream.read(inBuf,0,inBytes);
System.out.println(bytesRead+" bytes were read");
System.out.println(" Bytes read are: "+new String(inBuf));
inStream.close();
File f = new File("test.txt");
f.delete();
}
}
```

Output:

```
test.txt has 42 available bytes
42 bytes were read
Bytes read are: This program is for Testing I/O Operations.
```

Filtered Streams

One of the most powerful aspects of streams is that one stream can chain to the end of another. For example, the basic input stream only provides a read()method for reading bytes. If you want to read strings and integers, attach a special data input stream to an input stream and have methods for reading strings, integers, and even floats.

The FilterInputStream and FilterOutputStream classes provide the capability to chain streams together. The constructors for the FilterInputStream and FilterOutputStream take InputStream and OutputStream objects as parameters:

```
public FilterInputStream(InputStream in)
public FilterOutputStream(OutputStream out)
```

FilterInputStream has four filtering subclasses, -Buffer InputStream, Data

InputStream, LineNumberInputStream, and PushbackInputStream.
BufferedInputStream class: It maintains a buffer of the input data that it receives.

This eliminates the need to read from the stream's source every time an input byte is needed.

DataInputStream class: It implements the DataInput interface, a set of methods that allow objects and primitive data types to be read from a stream.

LineNumberInputStream class: This is used to keep track of input line numbers.

PushbackInputStream class: It provides the capability to push data back onto the stream that it is read from so that it can be read again.

The FilterOutputStream class provides three subclasses -BufferedOutputStream, DataOutputStream and Printstream.

BufferedOutputStream class: It is the output class analogous to the BufferedInputStream class. It buffers output so that output bytes can be written to devices in larger groups.

DataOutputStream class: It implements the DataOutput interface. It provides methods that write objects and primitive data types to streams so that they can be read by the DataInput interface methods.

PrintStream class: It provides the familiar print() and println() methods.

You can see in the program given below how objects of classes FileInputStream, FileOutputStream, BufferedInputStream, and BufferedOutputStream are used for I/O operations.

```
//program
import Java.io.*;
public class StreamsIODemo
{
  public static void main(String args[])
  {
  try
    {
    int a = 1;
    FileOutputStream fileout = new FileOutputStream("out.txt");
    BufferedOutputStream buffout = new BufferedOutputStream(fileout);
    while(a<=25)
    {
    buffout.write(a);
    a = a+3;
    }
    buffout.close();
    FileInputStream filein = new FileInputStream("out.txt");
    BufferedInputStream buffin = new BufferedInputStream(filein);
    int i=0;
    do
     {
    i=buffin.read();
    if (i!= -1)
    System.out.println(" "+ i);
    } while (i != -1) ;
buffin.close();
}
 catch (IOException e)
{
System.out.println("Eror Opening a file" + e);
}
}
}
```

Output:

```
1
4
7
10
13
16
19
22
25
```

2.3.2 Character Stream Classes

Character Streams are defined by using two class **Java.io.Reader and Java.io.Writer** hierarchies.

Both Reader and Writer are the abstract parent classes for character-stream based classes in the Java.io package. Reader classes are used to read 16-bit character streams and Writer classes are used to write to 16-bit character streams. The methods for reading from and writing to streams found in these two classes and their descendant classes (which we will discuss in the next section of this unit) given below:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Specialized Descendant Stream Classes

There are several specialized stream subclasses of the Reader and Writer class to provide additional functionality. For example, the BufferedReader not only provides buffered reading for efficiency but also provides methods such as "readLine()" to read a line from the input.

The following class hierarchy shows a few of the specialized classes in the Java.io package:

Reader

- BufferedReader
- LineNumberReader
- FilterReader
- PushbackReader
- InputStreamReader
- FileReader
- StringReader

Writer:

Now let us see a program to understand how the read and write methods can be used.

```
import Java.io.*;
public class ReadWriteDemo
  {
  public static void main(String args[]) throws Exception
  {
   FileReader fileread = new FileReader("StrCap.Java");
   PrintWriter printwrite = new PrintWriter(System.out, true);
   char c[] = new char[10];
   int read = 0;
   while ((read = fileread.read(c)) != -1)
     printwrite.write(c, 0, read);
   fileread.close();
   printwrite.close();
  }
  }
```

Output:

```
class StrCap
{
public static void main(String[] args)
{
StringBuffer StrB = new StringBuffer("Object Oriented Programming is possible in
Java");
String  Hi = new  String("This morning is very good");
System.out.println("Initial Capacity of StrB is :"+StrB.capacity());
System.out.println("Initial length of StrB is :"+StrB.length());
//System.out.println("value displayed by the expression Hi.capacity() is:
"+Hi.capacity());
System.out.println("value displayed by the expression Hi.length() is: "+Hi.length());
System.out.println("value displayed by the expression Hi.charAt() is:
"+Hi.charAt(10));
 }
}
```

Note: Output of this program is the content of file StrCap.Java.You can refer unit 3 of this block to see StrCap.Java file.

☞ Check Your Progress 1

1) What is stream? Differentiate between stream source and stream destination.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

2) Write a program for I/O operation using BufferedInputStream and BufferedOutputStream

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

3) Write a program using FileReader and PrintWriter classes for file handling.

…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………

2.4 THE PREDEFINED STREAMS

Java automatically imports the Java.lang package. This package defines a class called **System**, which encapsulates several aspects of run-time environment. **System** also contains three predefined stream objects, **in, out, and err.** These **objects** are declared as public and static within System. This means they can be used by other parts of your program without reference to your **System** object.

Access to standard input, standard output and standard error streams are provided via public static System.in, System.out and System.err objects. These are usually automatically associated with a user's keyboard and screen.

System.out refers to standard output stream. By default this is the console.
System.in refers to standard input, which is keyboard by default.
System.err refers to standard error stream, which also is console by default.
System.in is an object of InputStream. **System.out and System.err** are objects of **PrintStream.** These are byte streams, even though they typically are used to read and write characters from and to the console.

The predefined PrintStreams, out, and err within system class are useful for printing diagnostics when debugging Java programs and applets. The standard input stream System.in is available, for reading inputs.

2.5 READING FROM AND WRITING TO, CONSOLE

Now we will discuss how you can take input from console and see the output on console.

Reading Console Input

Java takes input from console by reading from System.in. It can be associated with these sources with reader and sink objects by wrapping them in a reader object. For System.in an InputStreamReader is appropriate. This can be further wrapped in a BufferedReader as given below, if a line-based input is required.
BufferedReader br = new BufferedReader (new InputStreamReader(System.in))

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

The program given below is for receiving console input in any of your Java applications**.**

```
//program
import Java.io.*;
class ConsoleInput
 {
  static String readLine()
   {
     StringBuffer response = new StringBuffer();
     try
```

```
        {
        BufferedInputStream buff = new BufferedInputStream(System.in);
        int in = 0;
        char inChar;
        do
        {
          in = buff.read();
          inChar = (char) in;
          if (in != -1)
          {
           response.append(inChar);
           }
          } while ((in != 1) & (inChar != '\n'));
          buff.close();
          return response.toString();
         }
        catch (IOException e)
           {
          System.out.println("Exception: " + e.getMessage());
          return null;
          }
    }
    public static void main(String[] arguments)
      {
       System.out.print("\nWhat is your name? ");
       String input = ConsoleInput.readLine();
       System.out.println("\nHello, " + input);
      }
      }
```

Output:

```
C:\JAVA\BIN>Java ConsoleInput

What is your name? Java Tutorial

Hello, Java Tutorial
```

Writing Console Output

You have to use System.out for standard Output in Java. It is mostly used for tracing the errors or for sample programs. These sources can be associated with a writer and sink objects by wrapping them in writer object. For standard output, an OutputStreamWriter object can be used, but this is often used to retain the functionality of print and println methods. In this case the appropriate writer is PrintWriter. The second argument to PrintWriter constructor requests that the output will be flushed whenever the println method is used. This avoids the need to write explicit calls to the flush method in order to cause the pending output to appear on the screen. PrintWriter is different from other input/output classes as it doesn't throw an IOException. It is necessary to send check Error message, which returns true if an error has occurred. One more side effect of this method is that it flushes the stream. You can create PrintWriter object as given below.

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

The ReadWriteDemo program discussed earlier in this section 2.3.2 of this unit is for reading and then displaying the content of a file. This program will give you an idea how to use FileReader and PrintWriter classes.

You may have observed that close()method is not required for objects created for standard input and output. You should have a question in mind–whether to use System.out or PrintWriter? There is nothing wrong in using System.out for sample programs but for real world applications PrintWriter is easier.

2.6 READING AND WRITING FILES

The streams are most often used for the standard input (the keyboard) and the standard output (theCRT display). Alternatively, input can arrive from a disk file using "input redirection", and output can be written to a disk file using "output redirection".

I/O redirection is convenient, but there are limitations to it. It is not possible to read data from a file using input redirection *and* receive user input from the keyboard at same time. Also, it is not possible to read or write multiple files using input redirection. A more flexible mechanism to read or write disk files is available in Java through its *file streams*.

Java has two file streams – the *file reader stream* and the *file writer stream*. Unlike the standard I/O streams, file stream must explicitly "open" the stream before using it.

Although, it is not necessary to close after operation is over, but it is a good practice to "close" the stream.

Reading Files

Let's begin with the FileReader class. As with keyboard input, it is most efficient to work through the BufferedReader class. If input is text to be read from a file, let us say "input.txt," it is opened as a file input stream as follows:

```
BufferedReader inputFile=new BufferedReader(new FileReader("input.txt"));
```

The line above opens, input.txt as a FileReader object and passes it to the constructor of the BufferedReader class. The result is a BufferedReader object named inputFile.

To read a line of text from input.txt, use the readLine() method of the BufferedReader class.

```
String s = inputFile.readLine();
```

You can see that input.txt is *not* being read using input redirection. It is explicitly opened as a file input stream. This means that the keyboard is still available for input. So, user can take the name of a file, instead of "hard coding".

Once you are finished with the operations on file, the file stream is closed as follows:
```
inputFile.close();
```

Some additional file I/O services are available through Java's File class, which supports simple operations with filenames and paths. For example, if fileName is a string containing the name of a file, the following code checks if the file exists and, if so, proceeds only if the user enters "y" to continue.

```
File f = new File(fileName);
if (f.exists())
{
System.out.print("File already exists. Overwrite (y/n)? ");
if(!stdin.readLine().toLowerCase().equals("y"))
return;
}
```

See the program written below open a text file called input.txt and to count the number of lines and characters in that file.

```
//program
import Java.io.*;
public class FileOperation
{
public static void main(String[] args) throws IOException
{
// the file must be called ‘input.txt’
String s = “input.txt”
File f = new File(s);
//check if file exists
if (!f.exists())
{
System.out.println("\'" + s + "\' does not exit!");
return;
}
// open disk file for input
BufferedReader inputFile = new BufferedReader(new FileReader(s));
// read lines from the disk file, compute stats
String line;
int nLines = 0;
int nCharacters = 0;
while ((line = inputFile.readLine()) != null)
{
nLines++;
nCharacters += line.length();
}
// output file statistics
System.out.println("File statistics for \'" + s + "\'...");
System.out.println("Number of lines = " + nLines);
System.out.println("Number of characters = " + nCharacters);
inputFile.close();
}
}
```

Output:

```
File statistics for ‘input.txt’…
Number of lines = 3
Number of characters = 7
```

Writing Files

You can open a file output stream to which text can be written. For this use the FileWriter class. As always, it is best to buffer the output. The following code sets up a buffered file writer stream named outFile to write text into a file named output.txt.

```
PrintWriter outFile = new PrintWriter(new BufferedWriter(new
FileWriter("output.txt"));
```

The object outFile, is an object of PrintWriter class, just like System.out. If a string, s, contains some text, to be written in “output.text”. It is written to the file as follows:
`outFile.println(s);`

When finished, the file is closed as:
`outFile.close();`

FileWriter constructor can be used with two arguments, where the second argument is a boolean type specifying an "append" option. For example, the expression new

FileWriter("output.txt", true) opens “output.txt” as a file output stream. If the file currently exists, subsequent output is appended to the file.

One more possibility is of opening an existing *read-only* file for writing. In this case, the program terminates with an “access is denied” exception. This should be caught and dealt within the program.

```
import Java.io.*;
class FileWriteDemo
{
public static void main(String[] args) throws IOException
{
// open keyboard for input
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
String s = "output.txt";
// check if output file exists
File f = new File(s);
if (f.exists())
{
  System.out.print("Overwrite " + s + " (y/n)? ");
   if(!stdin.readLine().toLowerCase().equals("y"))
   return;
 }
  // open file for output
 PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter(s)));
 System.out.println("Enter some text on the keyboard...");
 System.out.println("(^z to terminate)");
// read from keyboard, write to file output stream
String s2;
while ((s2 = stdin.readLine()) != null)
outFile.println(s2);
// close disk file
outFile.close();
}
}
```

Output:

```
Enter some text on the keyboard...
(^z to terminate)
hello students ! enjoying Java Session
^Z
```

Open out.txt you will find
“hello students ! enjoying Java Session” is stored in it.

☞ Check Your Progress 2

1) Which class may be used for reading from console?

………………………………………………………………………………………

………………………………………………………………………………………

2) Object of which class may be used for writing on console.

………………………………………………………………………………………

………………………………………………………………………………………

3) Write a program to read the output of a file and display it on console.

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

2.7 THE TRANSIENT AND VOLATILE MODIFIERS

Object serialization is very important aspect of I/O programming. Now we will discuss about the serializations.

Object serialization

It takes all the data attributes, writes them out as an object, and reads them back in as an object. For an object to be saved to a disk file it needs to be converted to a serial form. An object can be used with streams by implementing the serializable interface. The serialization is used to indicate that objects of that class can be saved and retrieved in serial form. Object serialization is quite useful when you need to use object persistence. By object persistence, the stored object continues to serve the purpose even when no Java program is running and stored information can be retrieved in a program so it can resume functioning unlike the other objects that cease to exist when object stops running.

DataOuputStreams and DataInputStreams are used to write each attribute out individually, and then can read them back in on the other end. But to deal with the entire object, not its individual attributes, store away an object or send it over a stream of objects. Object serialization takes the data members of an object and stores them away or retrieves them, or sends them over a stream.

ObjectInput interface is used for input, which extends the DataInput interface, and ObjectOutput interface is used for output, which extends DataOutput. You are still going to have the methods readInt(), writeInt() and so forth. ObjectInputStream, which implements ObjectInput, is going to read what ObjectOutputStream produces.

Working of object serialization

For ObjectInput and ObjectOutput interface, the class must be serializable. The serializable characteristic is assigned when a class is first defined. Your class must implement the serializable interface. This marker is an interface that says to the Java virtual machine that you want to allow this class to be serializable. You don't have to add any additional methods or anything.

There exists, a couple of other features of a serializable class. First, it has a zero parameter constructor. When you read the object, it needs to be able to construct and allocate memory for an object, and it is going to fill in that memory from what it has read from the serial stream. The static fields, or class attributes, are not saved because they are not part of an object.

If you do not want a data attribute to be serialized, you can make it transient. That would save on the amount of storage or transmission required to transmit an object. The transient indicates that the variable is not part of the persistent state of the object and will not be saved when the object is archived. Java defines two types of modifiers **Transient** and **Volatile.**

The volatile indicates that the variable is modified asynchronously by concurrently running threads.

Transient Keyword

When an object that can be serialized, you have to consider whether all the instance variables of the object will be saved or not. Sometimes you have some objects or sub objects which carry sensitive information like password. If you serialize such objects even if information (sensitive information) is private in that object if can be accessed from outside. To control this you can turn off serialization on a field- by-field basis using the transient keyword.

See the program given below to create a login object that keeps information about a login session. In case you want to store the login data, but without the password, the easiest way to do it is to implements **Serializable** and mark the **password** field as **transient**.

```
//Program
import Java.io.*;
import Java.util.*;
public class SerialDemo implements Serializable
{
  private Date date = new Date();
  private String username;
  private transient String password;
  SerialDemo(String name, String pwd)
  {
    username = name;
    password = pwd;
  }
  public String toString()
   {
    String pwd = (password == null) ? "(n/a)" : password;
    return "Logon info: \n   " + "Username: " + username +
      "\n   Date: " + date +  "\n   Password: " + pwd;
    }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException
    {
    SerialDemo a = new SerialDemo("Java", "sun");
    System.out.println( "Login is = " + a);
   ObjectOutputStream 0 = new ObjectOutputStream( new
   FileOutputStream("Login.out"));
   0.writeObject(a);
   0.close();
   // Delay:
   int seconds = 10;
   long t = System.currentTimeMillis()+ seconds * 1000;
   while(System.currentTimeMillis() < t)
      ;
   // Now get them back:
   ObjectInputStream in = new ObjectInputStream(new
   FileInputStream("Login.out"));
   System.out.println("Recovering object at " + new Date());
   a = (SerialDemo)in.readObject();
   System.out.println( "login a = " + a);
    }
}
```

Output:

```
Login is = Logon info:
Username: Java
Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: sun
Recovering object at Thu Feb 03 04:06:32 GMT+05:30 2005
login a = Logon info:
Username: Java
```

```
Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: (n/a)
```

In the above exercise Date and Username fields are ordinary (not **transient**), and thus are automatically serialized. However, the **password** is **transient**, and so it is not stored on the disk. Also the serialization mechanism makes no attempt to recover it. The **transient** keyword is for use with **Serializable** objects only.

Another example of transient variable is an object referring to a file or an input stream. Such an object must be created anew when it is part of a serialized object loaded from an object stream

```
// Donot serialize this field
private transient FileWriter outfile;
```

Volatile Modifier

The volatile modifier is used when you are working with multiple threads. The Java language allows threads that access shared variables to keep private working copies of the variables. This allows a more efficient implementation of multiple threads. These working copies need to be reconciled with the master copies in the shared (main) memory only at prescribed synchronization points, namely when objects are locked or unlocked. As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock and conventionally enforcing mutual exclusion for those shared variables. Only variables may be volatile. Declaring them so indicates that such methods may be modified asynchronously.

2.8 USING INSTANCE OF NATIVE METHODS

You will often feel the requirement that a Java application *must* communicate with the environment outside of Java. This is, perhaps, the main reason for the existence of native methods. The Java implementation needs to communicate with the underlying system – such as an operating system (as Solaris or Win32, or) a Web browser, custom hardware, (such as a PDA, Set-top-device,) etc. Regardless of the underlying system, there must be a mechanism in Java to communicate with that system. Native methods provide a simple clean approach to providing this interface between. Java and non-Java world without burdening the rest of the Java application with special knowledge.

Native Method is a method, which is not written in Java and is outside of the JVM in a library. This feature is not special to Java. Most languages provide some mechanism to call routines written in another language. In C++, you must use the extern "C" statement to signal that the C++ compiler is making a call to C functions.

To declare a native method in Java, a method is preceded with native modifiers much like you use the public or static modifiers, but don't define any body for the method simply place a semicolon in its place.

For example:

```
public native int meth();
```

The following class defines a variety of native methods:

```
public class IHaveNatives
{
native public void Native1(int x) ;
native static public long Native2();
native synchronized private float Native3( Object o ) ;
native void Native4(int[] ary) throws Exception ;
}
```

Native methods can be static methods, thus not requiring the creation of an object (or instance of a class). This is often convenient when using native methods to access an existing C-based library. Naturally, native methods can limit their visibility with the public, private, protected, or unspecified *default* access.

Every other Java method modifier can be used along with native, except *abstract*. This is logical, because the native modifier implies that an implementation exists, and the abstract modifier insists that there is no implementation.

The Following program is a simple demonstration of native method implementation.

```
class ShowMsgBox
 {
  public static void main(String [] args)
   {
    ShowMsgBox app = new ShowMsgBox();
    app.ShowMessage("Generated with Native Method");
    }
  private native void ShowMessage(String msg);
   {
    System.loadLibrary("MsgImpl");
   }
   }
```

☞ Check You Progress 3

1) What is Serialization?

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………..

……………………………………………………………………………….

2) Differentiate between Transient & Volatile keyword.

……………………………………………………………………………….

……………………………………………………………………………….

………………………………………………………………………………..

……………………………………………………………………………….

3) What are native method?

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………..

………………………………………………………………………………

………………………………………………………………………………

……………………………………………………………………………….

2.9 SUMMARY

This unit covers various methods of I/O streams-binary, character and object in Java. This unit briefs that input and output in the Java language is organised around the concept of streams. All input is done through subclasses of InputStream and all output is done through subclasses of OutputStream. (Except for RandomAccessFile). We have covered how various streams can be combined together to get the added functionality of standard input and stream input. In this unit you have also learned the operations of reading from a file and writing to a file. For this purpose objects of FileReader and FileWriter classes are used.

2.10 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) A stream is a sequence of bytes of undetermined length that travel from one place to another over a communication path.
Places from where the streams are picked–up are known as stream source. A source may be a file, input device or a network place-generating stream.
Places where streams are received or stored are known as stream destination. A stream destination may be a file, output device or a network place ready to receive stream.

2) This IO program is written using FileInputStream, BufferedInputStream , FileOutputStream, and BufferedOutputStream classes.

```
import java.io.*;
public class IOBuffer
{
  public static void main(String args[])
  {
  try
    {
    int x = 0;
    FileOutputStream FO = new FileOutputStream("test.txt");
    BufferedOutputStream BO = new BufferedOutputStream(FO);
    //Writing Data in BO
    while(x<=25)
    {
      BO.write(x);
       x = x+2;
     }
    BO.close();
    FileInputStream FI = new FileInputStream("test.txt");
    BufferedInputStream BI= new BufferedInputStream(FI);
    int i=0;
    do
    {
      i=BI.read();
      if (i!= -1)
      System.out.println(" " +i);
     } while (i != -1) ;
     BI.close();
     }
     catch (IOException e)
      {
```

```
                System.out.println("Eror Opening a file" + e);
               }
            }
        }
```

3) This program is written using FileReader and PrintWriter classes.

```
        import java.io.*;
public class FRPW
{
public static void main(String args[]) throws Exception
    {
    FileReader FR = new FileReader("Intro.txt");
    PrintWriter PW = new PrintWriter(System.out, true);
    char c[] = new char[10];
    int read = 0;
    while ((read = FR.read(c)) != -1)
    PW.write(c, 0, read);
    FR.close();
    PW.close();
    }
  }
```

Check Your Progress 2

1) InputStream class

2) PrintStream class

3) This program Reads the content from Intro.txt file and print it on console.

```
import java.io.*;
public class PrintConsol
{
  public static void main(String[] args)
  {
    try
     {
     FileInputStream FIS = new FileInputStream("Intro.txt");
      int n;
      while ((n = FIS.available()) > 0)
      {
      byte[] b = new byte[n];
      int result = FIS.read(b);
      if (result == -1) break;
      String s = new String(b);
      System.out.print(s);
       } // end while
      FIS.close();
     } // end try
     catch (IOException e)
     {
     System.err.println(e);
     }
    System.out.println();
   }
  }
```

Check Your Progress 3

1) Serialization is a way of implementation that makes objects of a class such that they are saved and retrieved in serial form. Serialization helps in making objects persistent.

2) Volatile indicates that concurrent running threads can modify the variable asynchronously. Volatile variables are used when multiple threads are doing the work. Transient keyword is used to declare those variables whose value need not persist when an object is stored.

3) Native methods are those methods, which are not written, in Java but they communicate to with Java applications to provide connectivity or to attach some systems such as OS, Web browsers, or PDA etc.

UNIT 3 STRINGS AND CHARACTERS

Structure **Page Nos.**

3.0 INTRODUCTION

In programming we use several data types as per our needs. If you observe throughout your problem solving study next to numeric data types character and strings are the most important data type that are used. In many of the programming languages strings are stored in arrays of characters (*e.g.* C, C++, Pascal, ...). However, in Java strings are a separate object type, called String. You are already using objects of String type in your programming exercises of previous units. A string is a set of character, such as "Hi". The Java platform contains three classes that you can use when working with character data: Character, String and StringBuffer. Java implements strings as object of String class when no change in the string is needed, and objects of StringBuffer class are used when there is need of manipulating of in the contents of string. In this unit we will discuss about different constructors, operations like concatenation of strings, comparison of strings, insertion in a string etc. You will also study about character extraction from a string, searching in a string, and conversion of different types of data into string form.

3.1 OBJECTIVES

After going through this unit you will be able to:

- explain Fundamentals of Characters and Strings;
- use different String and StringBuffer constructors;
- apply special string operations;
- extract characters from a string;
- perform such string searching & comparison of strings;
- data conversion using valueOf ( ) methods, and
- use StringBuffer class and its methods.

3.2 FUNDAMENTALS OF CHARACTERS AND STRINGS

Java provides three classes to deal with characters and strings. These are:

Character: Object of this class can hold a single character value. This class also defines some methods that can manipulate or check single-character data.

String: Objects of this class are used to deal with the strings, which are unchanging during processing.

String Buffer: Objects of this class are used for storing those strings which are expected to be manipulated during processing.

Characters

An object of Character type can contain only a single character value. You use a Character object instead of a primitive char variable when an object is required. For example, when passing a character value into a method that changes the value or when placing a character value into a Java defined data structure, which requires object. Vector is one of data structure that requires objects.

Now let us take an example program to see how Character objects are created and used.

In this program some character objects are created and it displays some information about them.

```
//Program
public class CharacterObj
{
public static void main(String args[])
{
Character Cob1 = new Character('a');
Character Cob2= new Character('b');
Character Cob3 = new Character('c');
int difference = Cob1.compareTo(Cob2);
if (difference == 0)
System.out.println(Cob1+"   is equal to  "+Cob2);
else
System.out.println(Cob1+"  is not equal to  "+Cob2);
System.out.println("Cob1 is " + ((Cob2.equals(Cob3)) ? "equal" : "not equal")+ " to
Cob3.");
}
}
```

Output:
```
a is not equal to b
Cob1 is not equal to Cob3.
```

In the above CharacterObj program following constructors and methods provided by the Character class are used.

Character (char): This is the Character class only constructor. It is used to create a Character object containing the value provided by the argument. But once a Character object has been created, the value it contains cannot be changed.

Compare to (Character): This instance method is used to compare the values held by two character objects. The value of the object on which the method is called and the value of the object passed as argument to the method.

For example, the statement

int difference = Cob1.compareTo(Cob2), in the above program.

This method returns an integer indicating whether the value in the current object is greater than, equal to, or less than the value held by the argument.

Equals (Character): This instance method is used to compare the value held by the current object with the value held by another (This method returns true if the values held by both objects are equal) object passed as on argument.

For example, in the statement

Cob2.equals(Cob3)
value held by object **Cob2** and the value held by **Cob3** will be compared.

In addition to the methods used in program CharacterObj, methods given below are also available in Character class.

To_String (): This instance method is used to convert the object to a string. The resulting string will have one character in it and contains the value held by the character object.

Char_Value(): An instance method that returns the value held by the character object as a primitive char value.

IsUpperCase (char): This method is used to determine whether a primitive char value is uppercase or not. It returns true if character is in upper case.

String and StringBuffer Classes

Java provides two classes for handling string values, which are **String** and

Can you tell why two String Classes?

The String class is provided for strings whose value will not change. For example, in program you write a method that requires string data and it is not going to modify the string.

The StringBuffer class is used for strings that will be modified. String buffers are generally used for constructing character data dynamically, for example, when you need to store information, which may change.

Content of strings do not change that is why they are more efficient to use than string buffers. So it is good to use String class wherever you need **fixed-length** objects that are **immutable (**size cannot be altered).

For example, the string contained by myString object given below cannot be changed.
String myString = "This content cannot be changed!".

☞ Check Your Progress 1

1) Write a program to find the case (upper or lower) of a character.

……………………………………………………………………………………

……………………………………………………………………………………

2) Explain the use of equal () method with the help of code statements.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) When should StringBuffer object be preferred over String object?

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

3.3 THE STRING CLASS

Now let us discuss String class in detail. There are many constructors provided in Java to create string objects. Some of them are given below.

public String(): Used to create a String object that represents an empty character sequence.

public String(String value): Used to create a String object that represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the string passed as an argument.

public String(char value[]): New object of string is created using the sequence of characters currently contained in the character array argument.

public String(char value[], int offset, int count): A new string object created contains characters from a sub-array of the character array argument.

public String(byte bytes[], String enc) throws Unsupported Encoding Exception: Used to construct a new String by converting the specified array of bytes using the specified character encoding.

public String(byte bytes[], int offset, int length): Used to create an object of String by converting the specified sub-array of bytes using the platform's default character encoding.

public String(byte bytes[]): Used to create an object of String by converting the specified array of bytes using the platform's default character encoding.

public String(StringBuffer buffer) : Used to create an object of String by using existing StringBuffer object which is passed as argument.

String class provides some important methods for examining individual characters of the strings, for comparing strings, for searching strings, for extracting sub-strings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

The Java language also provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

Note: When you create a String object, and, if it has the same **value** as another object, Java will point both object references to the same memory location.

3.4 STRING OPERATIONS

String class provides methods for handling of String objects. String class methods can be grouped in index methods, value Of () methods and sub-string methods. Methods of these groups are discussed below:

Index Methods

Methods	Return
public int length()	Return the length of string.
public int indexOf(int ch)	Return location of first occurrence of ch in string, if ch don’t exist in string return –1.
public int indexOf(int ch, int fromIndex)	Return location of occurrence of ch in string after fromIndex, if ch don’t exist in string return –1.
public int lastIndexOf(int ch)	Return location of last occurrence of ch in string, if ch location does not exist in string return –1.
public int lastIndexOf(int ch, int fromIndex)	Return last of occurrence of ch in string after location fromIndex, if ch does not exist in string after location fromIndex return –1.
public int indexOf(String str)	Return location of first occurrence of substring str in string, if str does not exist in string return –1.
public int indexOf(String str, int fromIndex)	Return location of first occurrence of substring str in after location fromIndex string, if str does not exist in string return –1.
public int lastIndexOf(String str)	Return location of last occurrence of substring str in string , if str does not exist in string return –1.
public int lastIndexOf(String str, int fromIndex)	Return location of last occurrence of substring str in after location from Index string, if str does not exist in string return –1.

You can see in the example program given below, in which the index methods are used. This program will help you to know the use of index methods.

```
public class Index_Methods_Demo
{
public static void main(String[] args)
{
String  str =  "This is a test string";
System.out.println(" The length of str is :"+ str.length());
System.out.println("Index of 't' in str is:"+str.indexOf('t'));
// Print first occurrence of character t in string
System.out.println("First occurrence of 't' after  13 characters in the str:"+
str.indexOf('t',12));
// Print first occurrence of character t in string after first 13 characters
System.out.println("Last occurrence of 'z' in the str:"+ str.lastIndexOf('z'));
// Print Last  occurrence of character z in string : See output and tell
// what is printed because 'z' is not a character in str.
System.out.println("First occurrence of substring :is substring of string  str:"+
str.indexOf("is"));
// Print first occurrence of substring "is" in str
System.out.println("Last occurrence of substring :ing in the str:"+
str.lastIndexOf("ing"));
// Print first occurrence of substring "ing" in string str
System.out.println("First occurrence of substring :this after  11 characters in the str:"+
str.indexOf("this",10));
// Print first occurrence of substring "this" after first 11 characters in string str
}
}
}
```

Output:

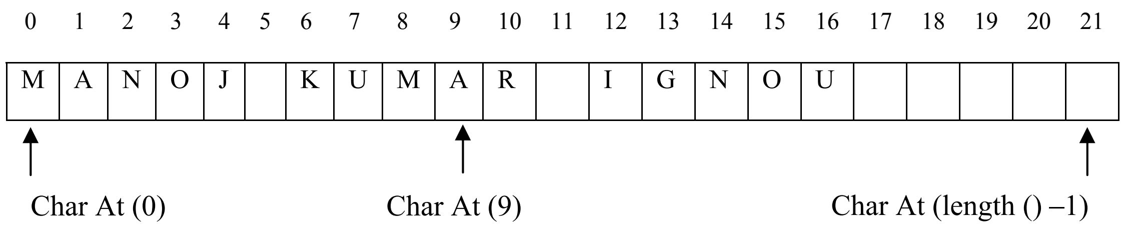
```
The length of str is :21
Index of 't' in str is:10
First occurrence of 't' after  13 characters in the str:13
Last occurrence of 'z' in the str:-1
First occurrence of substring :is substring of string  str:2
Last occurrence of substring :ing in the str:18
First occurrence of substring :this after  11 characters in the str:-1
```

In the output of the above given program you can see that if a character or substring does not exist in the given string then methods are returning: –1.

Now let us see some substring methods, comparisons methods, and string modifying methods provided in string class. These methods are used to find location of a character in string, get substrings, concatenation, comparison of strings, string reasons matching, case conversion etc.

Substring Methods

public char charAt(int index)	Return character at position i in string.



Methods	**Returns**
public byte[] getBytes()	Return byte array of string characters
public String substring(int beginIndex)	Return substring from index beginIndex to the end of string
public String substring	Return substring from index

(int beginIndex, int endIndex)	beginIndex to the endIndIndex of string
public String concat(String str)	Return a string which have sting on which this method is called Concatenated with str
public char[] toCharArray()	Return character array of string

Comparisons Methods

public boolean equals(Object str)	Return true if strings contain the same characters in the same order.
public boolean equalsIgnoreCase(String aString)	Return true if both the strings, - contain the same characters in the same order, and ignore case in comparison.
public int compareTo(String aString)	Compares to aString returns <0 if a String< sourceString 0 if both are equal and return >0 if aString>sourceString ; this method is case sensitive and used for knowing whether two strings are identical or not.
public boolean regionMatches (int toffset, String other, int offset, int len)	Return true if both the string have exactly same symbols in the given region.
public boolean startsWith (String prefix, int toffset)	Return true if prefix occurs at index toffset.
public boolean startsWith(String prefix)	Return true if string start with prefix.
public boolean endsWith(String suffix)	Return true if string ends with suffix.

Methods for Modifying Strings

public String replace(char oldChar, char newChar)	Return a new string with all oldChar replaced by newChar
public String toLowerCase()	Return anew string with all characters in lowercase
public String toUpperCase()	Return anew string with all characters in uppercase.
public String trim()	Return a new string with whitespace deleted from front and back of the string

In the program given below some of the string methods are used. This program will give you the basic idea about using of substring methods, comparisons methods, and string modifying methods.

```
public class StringResult
{
public static void main(String[] args)
{
String original = " Develop good software   ";
String sub1 = "";
String  sub2 = "Hi";
int index = original.indexOf('s');
sub1 =original.substring(index);
System.out.println("Substring sub1 contain: " + sub1);
System.out.println("Original contain: " + original+".");
original= original.trim();
System.out.println("After trim original contain: " + original+".");
System.out.println("Original contain: " + original.toUpperCase());
if (sub2.startsWith("H"))
System.out.println("Start with H: Yes");
else
System.out.println("Start with H: No");
sub2 = sub2.concat(sub1);
System.out.println("sub2 contents after concatenating sub1:"+sub2);
System.out.println("sub2 and  sub1 are equal:"+sub2.equals(sub1));
}
}
```

Output:

```
---------- Run ----------
Substring sub1 contain: software
Original contain:  Develop good software.
After trim original contain: Develop good software.
Original contain: DEVELOP GOOD SOFTWARE
Start with H: Yes
sub2 contents after concatenating sub1:Hisoftware
sub2 and  sub1 are equal:false
```

Now let us see valueOf() methods, these methods are used to convert different type of values like integer, float double etc. into string form. ValueOf () method is overloaded for all simple types and for Object type too. ValueOf() methods returns a string equivalent to the value which is passed in it as argument.

3.5 DATA CONVERSION USING VALUE OF( ) METHODS

public static String valueOf(boolean b)	Return string representation of the boolean argument
public static String valueOf(char c)	Return string representation of the character argument
public static String valueOf(int i)	Return string representation of the integer argument
public static String valueOf(long l)	Return string representation of the long argument
public static String valueOf(float f)	Return string representation of the float argument

public static String valueOf(double d)	Return string representation of the double argument
public static String valueOf(char data[])	Return string representation of the character array argument
public static String valueOf(char data[], offset, int count)	Return string representation of a int specific sub array of the character array argument

In the program given below you can see how an integer data is converted into a string type. This program also uses one very important operator '+' used for concatenating two strings.

```
class String_Test
{
public static void main(String[] args)
{
String s1 = new String("Your age is: ");
String s2 = new String();
int age = 28;
s2 = String.valueOf(age);
s1 = s1+s2+ " years";
System.out.println(s1);
}
}
```

Output:

```
Your age is: 28 years
```

☞ Check Your Progress 2

1) Write a program to find the length of string "Practice in programming is always good". Find the difference between first and last occurrence of 'r' in this string.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

2) Write a program which replaces all the occurrence of 'o' in string "Good Morning" with 'a' and print the resultant string.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

3) Write a program, which takes full path (directory path and file name) of a file and display Extension, Filename and Path separately for example, for input"/home/mem/index.html", output is Extension = html
Filename = index Path = /home/mem

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

You have seen that String class objects are of fixed length and no modification can be done in the content of strings except replacement of characters. If there is need of strings, which can be modified, then StrinBuffer class object should be used. Now let us discuss about StringBuffer class in detail.

3.6 STRINGBUFFER CLASS AND METHODS

StringBuffer: It is a peer class of String that provides much of the functionality of strings. In contrast to String objects, a StringBuffer object represents growable and writeable character sequences. In the strings created by using StringBuffer you may insert characters and sub-strings in the middle or append at the end. Three constructors of StringBuffer class are given below:

StringBuffer() Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

StringBuffer (int size) constructs a string buffer with no characters in it and an initial capacity specified by the length argument.

StringBuffer (String str) Constructs a string buffer so that it represents the same sequence of characters as the string argument; in other words, the initial contents of the string buffer is a copy of the argument string.

Methods of String Buffer Class

In addition to functionality of String class methods, SrtingBuffer also provide methods for modifying strings by inserting substring, deleting some content of string, appending string and altering string size dynamically. Some of the key methods in StringBuffer are mentioned below:

public int **length**()   Returns the length (number of characters)of the string buffer on which it is called.

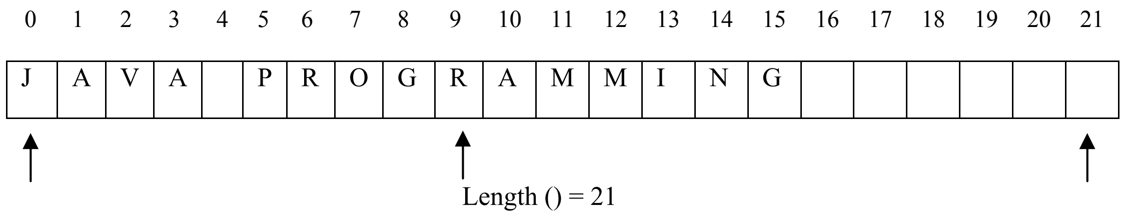


Figure 1: StringBuffer

public int **capacity**()	Returns the total allocated capacity of the String buffer. The capacity is the amount of storage available for characters can be inserted.
public void **ensureCapacity** (int minimumCapacity)	Used to ensures the capacity of the buffer is at least equal to the specified minimum capacity. If the current capacity of the string buffer on which this method is called is less than the argument value, then a new internal buffer is allocated with greater capacity and the new capacity is the larger than i) The minimumCapacity argument. ii) Twice the old capacity, plus 2.

	If you pass the minimumCapacity argument a nonpositive, value, then this method takes no action and simply returns.
public void **setLength**(int newLength)	Used to set the length of the string buffer on which it is called, if new length is less than the current length of the string buffer, the string buffer is truncated to contain exactly the number of characters given by the newLength argument. If the newLength argument is greater than or equal to the current length, string buffer is appended with sufficient null characters so that length becomes equal to the new Length argument. Obviously the newLength argument must be greater than or equal to 0.
public void **setCharAt**(int index, char ch)	Set character ch at the position specified by index in the string buffer on which it is called. This alters the character sequence that is identical to the old character sequence, except that now string buffer contain the character ch at position index and existing characters at index and after that in the string buffer are shifted right by one position. The value of index argument must be greater than or equal to 0, and less than the length of this string buffer.
public StringBuffer **append**(String newStr)	Appends the newStr to the string buffer. The characters of the string newStr are appended, to the contents of this string buffer increasing the length of the string buffer on which this method is called by the length of the newStr. Public.
StringBuffer **append** (StringBuffer strbuf)	Appends the characters of the strbuf to the string buffer, this results in increasing the length of the StringBuffer object on which this method is called. The method ensureCapacity is first called on this StringBuffer with the new buffer length as its argument (This ensures that the storage of this StringBuffer is adequate to contain the additional characters being appended).
public StringBuffer **append**(int i)	Appends the string representation of the int argument to the string buffer

	on which it is called. The argument is converted to a string by using method String.valueOf, and then the characters of converted string are then appended to the string buffer.
public StringBuffer **insert** (int offset, String str)	Insert the string str into the string buffer on which it is called at the indicated offset. It moves up characters originally after the offset position are shifted. Thus it increase the length of this string buffer by the length of the argument. The offset argument must be greater than or equal to 0, and less than or equal to the length of this string buffer.
public StringBuffer **insert** (int offset,char[] str)	Inserts the string representation of the array argument character into the string buffer on which this method is called. Insertion is made in the string buffer at the position indicated by offset. The length of this string buffer increases by the length of the argument
public StringBuffer **insert** ch (int offset, char ch)	Inserts the string representation of into the string buffer at the position indicated by offset. The length of this string buffer increases one.
public StringBuffer **reverse**()	Return the reverse of the sequence of the character sequence contained in the string buffer on which it is called. If n is the length of the old character sequence, in the string buffer just prior to execution of the reverse method. Then the character at index k in the new character sequence is equal to the character at index n-k-1 in the old character sequence.
public StringBuffer **delete**(int start, int end)	This method is used to remove total end–start +1 characters starting from location start and up to including index end.
public StringBuffer **deleteCharAt** (int index)	Removes the character at the specified position in the string buffer on which this method is called.
public StringBuffer **replace** (int start, int end, String str)	Replaces the characters of the string buffer from start to end by the characters in string str.

Now let us see use of some of the methods like capacity, append etc. of StringBuffer. As you know capacity method differs from length method, in that it returns the amount of space currently allocated for the StringBuffer, rather than the amount of space used. For example, in the program given below you can see that capacity of the StringBuffer in the reverseMe method never changes, while the length of the StringBuffer increases by one in each of the iteration of loop.

```
class ReversingString
  {
       public String reverseMe(String source)
       {
      int i, len = source.length();
      StringBuffer dest = new StringBuffer(len);
       System.out.println("Length of dest:"+dest.length());
       System.out.println("Capacity of dest:"+dest.capacity());
      for (i = (len - 1); i >= 0; i--)
      dest.append(source.charAt(i));
       System.out.println("Capacity of dest:"+dest.capacity());
       System.out.println("Length of dest:"+dest.length());
      return dest.toString();
     }
     }
       public class ReverseString
       {
       public static void main ( String args[])
       {
       ReversingString R = new RString();
       String myName = new String("Mangala");
       System.out.println("Length of myName:"+myName.length());
         System.out.println(" myName:"+myName);
       System.out.println("reverseMe call:"+R.reverseMe(myName));
       }
```

```
Output:
Length of myName:7
myName:Mangala
Length of dest:0
Capacity of dest:7
Capacity of dest:7
Length of dest:7
Reverse call:alagnaM
```

Note:

i. In the reverseMe () method of above StringBuffer's to String () method is used to convert the StringBuffer to a String object before returning the String.

ii. You should initialize a StringBuffer's capacity to a reasonable first guess, because it minimizes the number of times memory to be allocated for the situation when the appended character causes the size of the StringBuffer to grow beyond its current capacity. Because memory allocation is a relatively expensive operation, you can make your code more efficient by initializing a StringBuffer's capacity to a reasonable first guess size.

Now let us take one more example program to see how to insert data into the middle of a StringBuffer. This is done with the help of one of StringBufffer's insert methods.

```
class  InsertTest
{
public static void main ( String args[])
{
StringBuffer MyBuffer = new StringBuffer("I got class in B.Sc.");
MyBuffer.insert(6, "First ");
System.out.println(MyBuffer.toString());
}
}
```
Output:
```
I got First class in B.Sc.
```

☞ Check Your Progress 3

1) List any two operations that you can perform on object StringBuffer but cannot perform on object of String class.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) Write a program to answer the following:

   i. Find the initial capacity and length of the following string buffer:
   StringBuffer StrB = new StringBuffer(“Object Oriented Programming is possible in Java”);

   ii. Consider the following string:

   String Hi = “This morning is very good”;
   What is the value displayed by the expression Hi.capacity()?
   What is the value displayed by the expression Hi.length()?
   What is the value returned by the method call Hi.charAt(10)?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) Write a program that finds initials from your full name and displays them.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3.7 SUMMARY

Character and strings are the most important data type. Java provides Character, String, and StringBuffer classes for handling characters, and strings. String class is used for creating the objects which are not to be changed. The string objects for which contents are to be changed StringBuffer class, is used. Character class provides various methods like compareTo, equals, isUpperCase.String class provides methods for index operations, substring operations, and a very special group of valueOf methods. Comparison methods are also provided in String class.String class methods like replace, toLowerCase, trim are available for minor modifications though, in general string classes is not used for dynamic Strings. StringBuffer objects allow to insert character or substring in the middle or append it to the end. StringBuffer also allows deletion of a substring of a string.

3.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1)
```
//Program to test whether the content of a Character object is Upper or Lower
public class CharTest
{
public static void main(String args[])
{
Character MyChar1 = new Character('i');
Character MyChar2 = new Character('J');
//Test for MyChar1
if (MyChar1.isUpperCase(MyChar1.charValue()))
System.out.println("MyChar1 is in Upper Case: "+MyChar1);
else
System.out.println("MyChar1 is in Lower Case: "+MyChar1);
// Test for MyChar2
if (MyChar2.isUpperCase(MyChar2.charValue()))
System.out.println("MyChar2 is in Upper Case: "+MyChar2);
else
System.out.println("MyChar2 is in Lower Case: "+MyChar2);

}
}
```
Output:
```
MyChar1 is in Lower Case: i
MyChar2 is in Upper Case: J
```

2) This instance method is used to compare the value held by the current object with the value held by another object. For example let Ch1 and Ch2 be two objects of Character class then
Ch1.equals(Ch2);
method returns true if the values held by both objects are equal, otherwise false.

3) StringBuffer object should be given preference over String objects if there is a need of modification (change may take place) in the content. These include flexibility of increase in size of object.

Check Your Progress 2

1)
```
class StringLen
{
public static void main(String[] args)
{
String MyStr = new String(" Practice in programming is always good");
System.out.println("The length of MyStr is :"+MyStr.length());
int i = MyStr.lastIndexOf("r")- MyStr.indexOf("r");
System.out.println("Difference between first and last occurence of 'r' in MyStr
is :"+ i);

}
}
Output:
The length of MyStr is: 39
Difference between first and last occurence of 'r' in MyStr is: 15
```

2)
```
        //program
        class ReplaceStr
        {
        public static void main(String[] args)
        {
        String S1 = new String("Good Morning");
        System.out.println("The old String is:"+S1);
        String S2= S1.replace('o', 'a');
        System.out.println("The new String after rplacement of 'o' with 'a' is:"+S2);
        }
        }
```

Output:

```
The old String is:Good Morning
The new String after replacement of 'o' with 'a' is: Good Marning
```

3)
```
      // It is assumed that  fullPath has a directory path, filename, and extension.
      class Filename
 {
   private String fullPath;
   private char pathSeparator, extensionSeparator;
   public Filename(String str, char separ, char ext)
   {
      fullPath = str;
      pathSeparator = separ;
      extensionSeparator = ext;
   }
   public String extension()
   {
      int dot = fullPath.lastIndexOf(extensionSeparator);
      return fullPath.substring(dot + 1);
    }
   public String filename()
   {
      int dot = fullPath.lastIndexOf(extensionSeparator);
      int separ = fullPath.lastIndexOf(pathSeparator);
      return fullPath.substring(separ + 1, dot);
   }
   public String path()
   {
      int separ = fullPath.lastIndexOf(pathSeparator);
      return fullPath.substring(0, separ);
   }
}
// Main method  is in FilenameDemo class
  public class FilenameDemo1
  {
   public static void main(String[] args)
   {
      Filename myHomePage = new Filename("/HomeDir/MyDir/MyFile.txt",'/', '.');
      System.out.println("Extension = " + myHomePage.extension());
      System.out.println("Filename = " + myHomePage.filename());
      System.out.println("Path = " + myHomePage.path());
    }
}
```

Output:

```
Extension = txt
Filename = MyFile
Path = /HomeDir/MyDir
```

Note:

In this program you can notice that extension uses dot + 1 as the argument to substring. If the period character is the last character of the string, then dot + 1 is equal to the length of the string which is one larger than the largest index into the string (because indices start at 0). However, substring accepts an index equal to but not greater than the length of the string and interprets it to mean “the end of the string.”

Check Your Progress 3

1) i. Insertion of a substring in the middle of a string.

   ii. Reverse the content of a string object.

2)

```
class StrCap
{
public static void main(String[] args)
{
StringBuffer StrB = new StringBuffer("Object Oriented Programming is
possible in Java");
String  Hi = new  String("This morning is very good");
System.out.println("Initial Capacity of StrB is :"+StrB.capacity());
System.out.println("Initial length of StrB is :"+StrB.length());

//System.out.println("value displayed by the expression Hi.capacity() is:
"+Hi.capacity());
System.out.println("value displayed by the expression Hi.length() is:
"+Hi.length());
System.out.println("value displayed by the expression Hi.charAt() is:
"+Hi.charAt(10));
}
}
```

Output:

```
Initial Capacity of StrB is :63
Initial length of StrB is :47
value displayed by the expression Hi.length() is: 25
value displayed by the expression Hi.charAt() is: n
```

Note: The statement “System.out.println("value displayed by the expression Hi.capacity() is: "+Hi.capacity());” is commented for successful execution of program because capacity method is mot available in String class.

3)

```
public class NameInitials
{
public static void main(String[] args)
{
String myNameIs = "Mangala Prasad Mishra";
StringBuffer myNameInitials = new StringBuffer();
System.out.println("The name is : "+myNameIs);
// Find length of name given
int len = myNameIs.length();
```

```
          for (int i = 0; i < len; i++)
                                {
          if (Character.isUpperCase(myNameIs.charAt(i)))
                                   {
          myNameInitials.append(myNameIs.charAt(i));
         }
         }
         System.out.println("Initials of the name: " + myNameInitials);
         }
         }
```

Output:

```
The name is: Mangala Prasad Mishra
Initials of the name: MPM
```

UNIT 4 EXPLORING JAVA I/O

Structure **Page Nos.**

4.0 INTRODUCTION

Input is any information that is needed by your program to complete its execution and *Output* is information that the program must produce after its execution. Often a program needs to bring information from an external source or to send out information to an external destination. The information can be anywhere in file. On disk, somewhere on network, in memory or in other programs. Also the information can be of any type, i.e., object, characters, images or sounds. In Java information can be stored, and retrieved using a communication system called streams, which are implemented in Java.io package.

The Input/output occurs at the program interface to the outside world. The input-output facilities must accommodate for potentially wide variety of operating systems and hardware. It is essential to address the following points:

- Different operating systems may have different convention how an end of line is indicated in a file.
- The same operating system might use different character set encoding.
- File naming and path name conventions vary from system to system.

Java designers have isolated programmers from many of these differences through the provision of package of input-output classes, java.io. Where data is stored objects of the reader classes of the java.io package take data, read from the files according to convention of host operating system and automatically converting it.

In this unit we will discuss how to work on streams. The dicussion will be in two directions: pulling data into a program over an input stream and sending data from a program using an output stream.

In this unit you will work with I/O classes and interfaces, streams File classes, Stream tokenizer, Buffered Stream, Character Streams, Print Streams, and Random Access Files.

4.1 OBJECTIVES

After going through this unit you will be able to:

- explain Java I/O hierarchy;

- handle files and directories using File class;
- read and write using I/O stream classes;
- differentiate between byte stream, character stream and text stream;
- use stream tokenizer in problem solving;
- use Print stream objects for print operations, and
- explain handling of random access files.

4.2 JAVA I/O CLASSES AND INTERFACES

The package java.io provides two sets of class hierarchies-one for reading and writing of bytes, and the other for reading and writing of characters. The InputStream and OutputStream class hierarchies are for reading and writing bytes. Java provides class hierarchies derived from Reader and Writer classes for reading and writing characters.

A character encoding is a scheme for internal representation of characters.

Java programs use 16 bit Unicode character encoding to represent characters internally. Other platforms may use a different character set (for example ASCII) to represent characters. The reader classes support conversions of Unicode characters to internal character storage.

Classes of the j*ava.io* hierarchy

Hierarchy of classes of Java.io package is given below:

- ♦ InputStream
  - FilterInputStream
    - ▪ BufferedInputStream
    - ▪ DataInputStream
    - ▪ LineNumberInputStream
    - ▪ PushbackInputStream
  - ByteArrayInputStream
  - FileInputStream
  - ObjectInputStream
  - PipedInputStream
  - SequenceInputStream
  - StringBufferInputStream
- ♦ OutputStream
  - FilterOutputStream
    - ▪ BufferedOutputStream
    - ▪ DataOutputStream
    - ▪ PrintStream
  - ByteArrayOutputStream
  - FileOutputStream
  - ObjectOutputStream
  - PipedOutputStream
- ♦ Reader
  - BufferedReader
    - ▪ LineNumberReader
  - CharArrayReader
  - FilterReader
    - ▪ PushbackReader
  - InputStreamReader
    - ▪ FileReader
  - PipedReader
  - StringReader

- Writer
    - BufferedWriter
    - CharArrayWriter
    - FilterWriter
    - OutputStreamWriter
        - FileWriter
    - PipedWriter
    - PrintWriter
    - StringWriter
- File
    - RandomAccessFile
    - FileDescriptor
    - FilePermission
    - ObjectStreamClass
    - ObjectStreamField
    - SerializablePermission
    - StreamTokenizer.

Interfaces of Java.io

Interfaces in Java.io are given below:

- DataInput
- DataOutput
- Externalizable
- FileFilter
- FilenameFilter
- ObjectInput
- ObjectInputValidation
- ObjectOutput
- ObjectStreamConstants
- Serializable

Each of the Java I/O classes is meant to do one job and to be used in combination with other to do complex tasks. For example, a BufferedReader provides buffering of input, nothing else. A FileReader provides a way to connect to a file. Using them together, you can do buffered reading from a file. If you want to keep track of line numbers while reading from a character file, then by combining the File Reader with a LineNumberReader to serve the purpose.

4.3 I/O STREAM CLASSES

I/O classes are used to get input from any source of data or to send output to any destination. The source and destination of input and output can be file or even memory. In Java input and output is defined in terms of an abstract concept called stream. A Stream is a sequence of data. If it is an input stream it has a source. If it is an output stream it has a destination. There are two kinds of streams: byte stream and character stream. The java.io package provides a large number of classes to perform stream IO.

The File Class

The File class is not I/O. It provides just an identifier of files and directories. Files and directories are accessed and manipulated through java.io.file class. So, you always remember that creating an instance of the File class does not create a file in the operating system. The object of the File class can be created using the following types of File constructors:

```
File MyFile = new File("c:/Java/file_Name.txt");
File MyFile = new File("c:/Java", "file_Name.txt");
File MyFile = new File("Java", "file_Name.txt");
```

The first type of constructor accepts only one string parameter, which includes file path and file name, here in given format the file name is file_Name.txt and its path is c:/Java.

In the second type, the first parameter specifies directory path and the second parameter denotes the file name. Finally in the third constructor the first parameter is only the directory name and the other is file name.
Now let us see the methods of the file class.

Methods

boolean exists() : Return true if file already exists.
boolean canWrite() : Return true if file is writable.
boolean canRead() : Return true if file is readable.
boolean isFile() : Return true if reference is a file and false for directories references.
boolean isDirectory(): Return true if reference is a directory.
String getAbsolutePath() : Return the absolute path to the application

You can see the program given below for creating a file reference.

```
//program
import java.io.File;
class FileDemo
        {
        public static void main(String args[])
        {
        File f1 = new File ("/testfile.txt");
        System.out.println("File name : " + f1.getName());
        System.out.println("Path :  " + f1.getPath());
        System.out.println("Absolute Path : " + f1.getAbsolutePath());
        System.out.println(f1.exists() ? "Exists" : "doesnot exist");
        System.out.println( f1.canWrite()?"Is writable" : "Is not writable");
        System.out.println(f1.canRead() ? "Is readable" :"Is not readable");
        System.out.println("File Size : " + f1.length() + " bytes");
        }
    }
```

Output: (When testfile.txt does not exist)

```
File name: testfile.txt
Path:  \testfile.txt
Absolute Path: C:\testfile.txt
doesnot exist
Is not writable
Is not readable
File Size: 0 bytes
```

Output: (When testfile.txt exists)

```
File name : testfile.txt
Path :  \testfile.txt
Absolute Path : C:\testfile.txt
Exists
Is writable
Is readable
File Size: 17 bytes
```

4.3.1 Input and Output Stream

Java Stream based I/O builts upon four abstract classes: InputStream, OutputStream, Reader and Writer. An object from which we can *read a sequence of bytes* is called an *input stream*. An object from which we can *write sequence of byte* is called *output stream*. Since byte oriented streams are inconvenient for processing. Information is stored in Unicode characters that inherit from abstract *Reader and Writer* superclasses.

All of the streams: The readers, writers, input streams, and output streams are automatically opened when created. You can close any stream explicitly by calling its close method. The garbage collector implicitly closes it if you don't close. It is done when the object is no longer referenced. Also, the *direction of flow* and *type* of data is not the matter of concern; algorithms for sequentially reading and writing data are basically the same.

Reading and Writing IO Stream

Reader and InputStream define similar APIs but for different data types. You will find, that both Reader and InputStream provide methods for marking a location in the stream, skipping input, and resetting the current position. *For example,* Reader and Input Stream defines the methods for reading characters and arrays of characters and reading bytes and array of bytes respectively.

Methods

int read() : reads one byte and returns the byte that was read, or –1 if it encounters the end of input source.

int read(char chrbuf[]): reads into array of bytes and returns number of bytes read or –1 at the end of stream.
int read(char chrbuf[], int offset, int length): chrbuf – the array into which the data is read. offset - is offset into chrbuf where the first byte should be placed. len - maximum number of bytes to read.

Writer and OutputStream also work similarly. Writer defines these methods for writing characters and arrays of characters, and OutputStream defines the same methods but for bytes:

Methods

int write(int c): writes a byte of data
int write(char chrbuf[]) : writes character array of data.
int write(byte chrbuf[]) writes all bytes into array b.
int write(char chrbuf[], int offset, int length) : Write a portion of an array of characters.

☞ **Check Your Progress 1**

1) What is Unicode? What is its importance in Java?

……………………………………………………………………………………

……………………………………………………………………………………

2) Write a program which calculates the size of a given file and then renames that file to another name.

……………………………………………………………………………………

……………………………………………………………………………………

3) Write a program which reads string from input device and prints back the same on the screen.

……………………………………………………………………………

……………………………………………………………………………

4.3.2 Input Stream and Output Stream hierarchy

There are two different types of Streams found in Java.io package as shown in Figure 1a OutputStream and InputStream. Figure 1b shows the further classification of Inputstream.

The following classes are derived from InputStream Class.

InputStream: Basic input stream.
StringBufferInputStream: An input stream whose source is a string.
ByteArrayInputStream: An input stream whose source is a byte array.
FileInputStream: An input stream used for basic file input.

FilterInputStream: An abstract input stream used to add new behaviour to existing input stream classes.

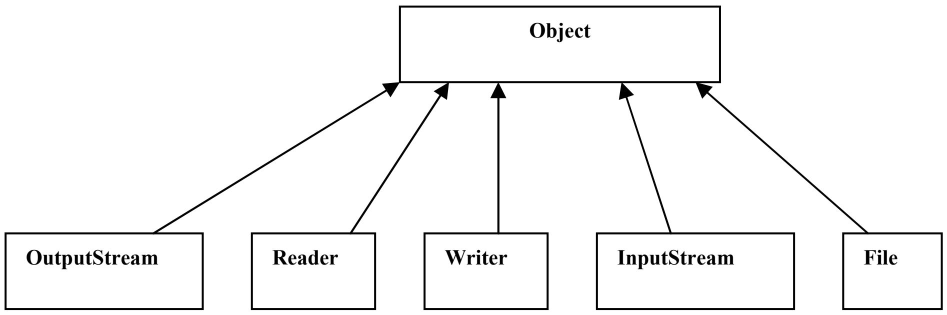


Figure 1(a): Types of stream in Java I/O

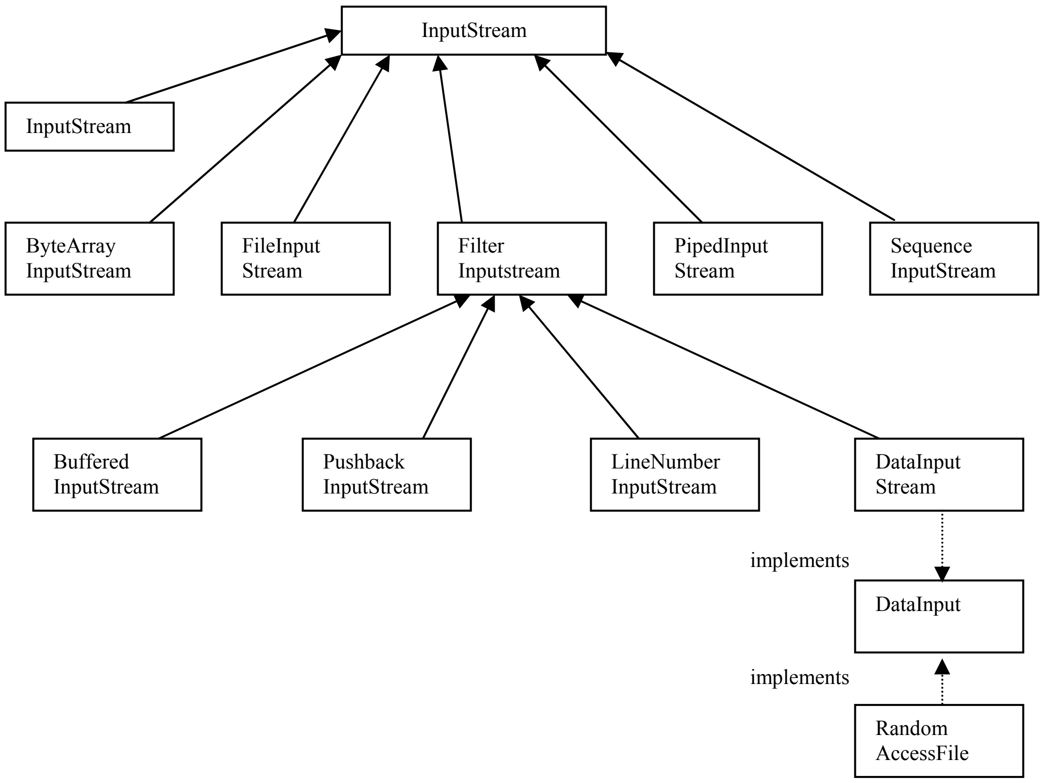


Figure 1(b): Input Stream Class

PipedInputStream: An input stream used for inter-thread communication.
SequenceInputStream: An input stream that combines two other input streams.
BufferedInputStream: A basic buffered input stream.
PushbackInputStream: An input stream that allows a byte to be pushed back onto the stream after the byte is read.
LineNumberInputStream: An input stream that supports line numbers.
ataInputStream: An input stream for reading primitive data types.
RandomAccessFile: A class that encapsulates a random access disk file.

Output streams are the logical counterparts to input streams and handle writing data to output sources. *In Figure2* the hierarchy of output Stream is given. The following classes are derived from outputstream.

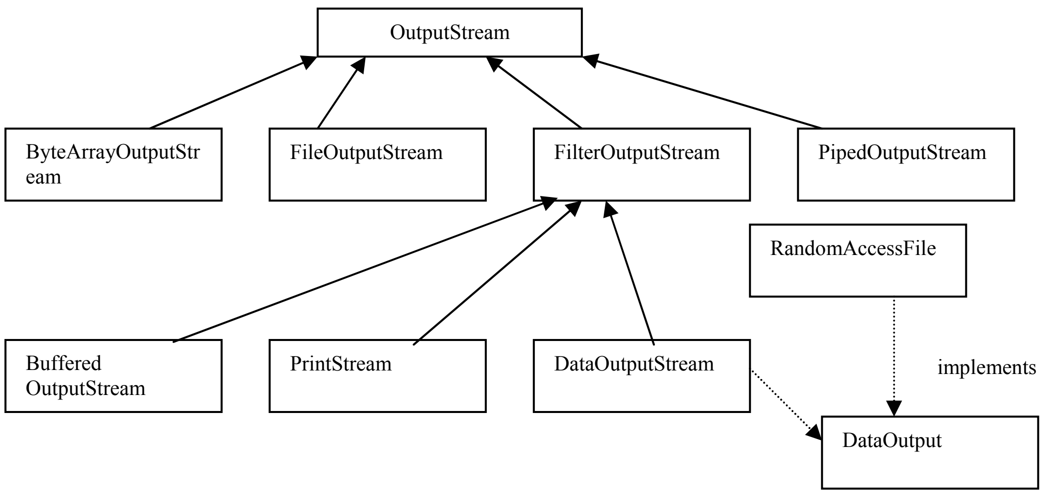


Figure 2: Output Stream Class Hierarchy

OutputStream: The basic output stream.
ByteArrayOutputStream: An output stream whose destination is a byte array.
FileOutputStream: Output stream used for basic file output.
PipedOutputStream: An output stream used for inter-thread communication.
BufferedOutputStream: A basic buffered output stream.
PrintStream: An output stream for displaying text.
DataOutputStream: An output stream for writing primitive data types.
FilterOutputStream: An abstract output stream used to add new behaviour to existing output stream classes.

Layering byte Stream Filter

Some-times you will need to combine the two input streams. in Java. It is known as **filtered streams** *(i.e., feeding an existing stream to the constructor of another stream)*. You should continue layering stream constructor until you have access to the functionality you want.

FileInputStream and FileOutputStream give you input and output streams attached to a disk file. For example giving file name or full path name of the file in a constructor as given below:

FileInputStream fin = new FileInputStream("Mydat.dat");

Input and output stream classes only support reading and writing on byte level, DataInputStream has a method that could read numeric. FileInputStream has no method to read numeric type and the DataInputStream has no method to get data from the file. If you have to read the numbers from file, first create a FileInputStream and pass it to DataInputStream, as you can see in the following code:

FileInputStream fin = new FileInputStream ("Mydat.dat");

```
DataInputStream din = new DataInputStream (fin)
doubles = din.readDouble();
```

If you want buffering, and data input from a file named Mydata.dat then write your code like:

```
DataInputStream din = new DataInputStream(new BufferedInputStream
                                        (new FileInputStream("employee.dat")));
```

Now let us take one example program in which we open a file with the binary FileOutputStream class. Then wrap it with DataOutput Stream class. Finally write some data into this file.

```
//program
import java.io.*;
public class BinFile_Test
{
    public static void main(String args[])
    {
    //data array
    double data [] = {1.3,1.6,2.1,3.3,4.8,5.6,6.1,7.9,8.2,9.9};
    File file = null;
    if (args.length > 0 ) file = new File(args[0]);
    if ((file == null) || !(file.exists()))
    {
    // new file created
    file = new File("numerical.txt");
    }
    try
    {
     // Wrap the FileOutput Stream with DataOutputStream to obtain its writeInt()
     method
    FileOutputStream fileOutput = new FileOutputStream(file);
    DataOutputStream dataOut    = new DataOutputStream(fileOutput);
    for (int i=0; i < data.length;i++)
    dataOut.writeDouble(data[i]);
    }
    catch (IOException e)
    {
     System.out.println("IO error is there " + e);
    }
    }
    }
```

Output:

You can see numeric.txt created in your system you will find something like:

```
?ôÌÌÌÌÍ?ù™™™™™š@
```

Reading from a Binary File

Similarly, we can read data from a binary file by opening the file with a **FileInputStream** Object.

Then we wrap this with a **DataInputStream class** to obtain the many **readXxx()** methods that are very useful for reading the various primitive data types.

```
import java.io.*;
public class BinInputFile_Appl
```

```
{
  public static void main(String args[])
   {
    File file = null;
    if (args.length > 0 ) file = new File(args[0]);
    if ((file == null) || !file.exists())
    {
     file = new File("numerical.dat");
     }
     try
     {   // Wrap the FileOutput Stream with DataInputStream so that we can use its
       readDouble() method
       FileInputStream fileinput = new FileInputStream(file);
       DataInputStream dataIn    = new DataInputStream(fileinput);
        int i=1;
       while(true)
       {
         double d = dataIn.readDouble();
         System.out.println(i+".data="+d);
        }
      }
      catch (EOFException eof)
       {
        System.out.println("EOF Reached " + eof);
        }
      catch (IOException e)
       {
        System.out.println(" IO erro" + e);
        }
        }
        }
```

Output:

```
C:\JAVA\BIN>Java BinInputFile_Appl
1.data=1.3
1.data=1.6
1.data=2.1
1.data=3.3
1.data=4.8
1.data=5.6
1.data=6.1
1.data=7.9
1.data=8.2
1.data=9.9
```

Reading/Writing Arrays of Bytes

Some time you need to read/write some bytes from a file or from some network place. For this purpose classes Bytes Array InputStream and ByteArrayOutputStream are available.

Constructors of ByteArrayInputStream

ByteArrayInputStream(byte[] input_buf, int offset, int length)
ByteArrayInputStream(byte[] input_buf)
In the first constructor input_buf is the input buffer, offset indicates the position of the first byte to read from the buffer and length tells about the maximum number of bytes to be read from the buffer. Similarly for simple purpose another constructor is defined which uses input_buf its buffer array and also contain information to read this array.

For example in the code given below ByteArrayInputStream creates two objects input1 and input2, where input1 contains all letters while input2 will contain only first three letters of input stream ("abc").

```
String tmp = "abcdefghijklmnopqrstuvwxyz"
byte b[] =  tmp.getBytes();
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
```

ByteArrayOutputStream is an output stream that is used to write byte array. This class has two constructors- one with no parameter and another with an integer parameter that tells the initial size of output array.

```
ByteArrayOutputStream( )
ByteArrayOutputStream( int buf_length)
```

In the following piece of code you can see how ByteArrayOutputStream provides an output with an array of bytes.

```
        ByteArrayOutputStream f = new ByteArrayOutputStream();
        String tmp = "abcdefghijklmnopqrstuvwxyz"
        byte b[] =  tmp.getBytes();
        f,write(b);
```

PushbackInput Stream is an input stream that can unwrite the last byte read

You often need to peek at the next byte to see whether it is a value that you expect. Use **PushbackInputStream** for this purpose. You can use this class to unwrite the last byte read is given in the following piece of code:

```
PushbackInputStream pbin = new PushbackInputStream
(new BufferedInputStream(new FileInputStream ("employee.dat")));
```

You can read the next byte, as follow

```
            int b = pbin.read();
                if (b != '< ') pbin.read.unread(b);
```

Read() and Unread() are the only methods that apply to pushback input stream. If you want to look ahead and also read numbers, you need to combine both a pushback inputstream and a data. Input. stream reference.

```
        DatainputStream din = new DataInputStream
(PushBackInputStream pbin = new PushBackInputStream
(new bufferedInputStream (newFileInputStream("employee.dat"))));
```

4.4 TEXT STREAMS

Binary IO is fast but it is not readable by humans. For example, if integer 1234 is saved as binary, it is written as a sequence of bytes **00 00 04 D2** (in hexadecimal notation). In text format it is saved as the string "1234". Java uses its own character-encoding. This may be single byte, double byte or a variable byte scheme based on host system. If the Unicode encoding is written in text file, then the resulting file will unlikely be human readable. To overcome this Java has a set of stream filters that bridges the gap between Unicode encoded text and character encoding used by host. All these classes are descended from abstract *Reader and Writer classes.*

Reader and Writer classes

You may know that Reader and Writer are abstract classes that define the java model of streaming characters. These classes were introduced to support 16 bit Unicode characters in java 1.1 and onwards and provide Reader/Writer for java 1.0 stream classes.

java 1.0	**java 1.1**
InputStream	Reader
OutputStream	Writer
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

For *filter classes,* we also have corresponding *Reader* and *Writer* classes as given below:

FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract)
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
DataInputStream	DataInputStream
	BuffedReader (readLine())
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer
PushBackInputStream	PushBackReader

In addition to these classes java 1.1 also provides two extra classes which provide a bridge between byte and character streams. For reading bytes and translating them into characters according to specified character encoding we have InputStreamReader while for the reverse operation translating characters into bytes according to a specified character encoding java 1.1 provides OutputStreamWriter.

Reading Text from a File

Let us see how to use a FileReader stream to read character data from text files. In the following example read () method obtains one character at a time from the file. We are reading bytes and counting them till it meets the EOF (end of file). Here FileReader is used to connect to a file that will be used for input.

```
// Program
import java.io.*;
public class TextFileInput_Appl
{
public static void main(String args[])
{
File file = null;
        if (args.length > 0 ) file= new File(args[0]);
        if (file == null || !file.exists())
        {
        file=new File("TextFileInput_Appl.Java");
        }
        int numBytesRead=0;
        try
{
int tmp =0 ;
FileReader filereader = new FileReader (file);
while( tmp != -1)
{
tmp= filereader.read();
if (tmp != -1) numBytesRead++;
}
}
catch (IOException  e)
{
System.out.println(" IO erro" + e);
}
System.out.println(" Number of bytes read : " + numBytesRead);
System.out.println(" File.length()= "+file.length());
}
}
```

For checking whether we read all the bytes of the file “TextFileInput_Appl.Java” in this program we are comparing both, length of file and the bytes read by read( ) method. You can see and verify the result that both values are 656.

Output:

```
Number of bytes read: 656
File.length()= 656
```

Writing Text to a file

In the last example we read bytes from a file. Here let us write some data into file. For this purpose Java provides FileWriter class. In the following example we are demonstrating you the use of FileWriter. In this program we are accepting the input data from the keyboard and writes to a file a "textOutput.txt". To stop writing in file you can press ctrl+C which will close the file.

```
// Program Start here
import java.io.*;
public class TextFileOutput_Appl
{
  public static void main(String args[])
  {
    File file = null;
    file = new File("textOutput.txt");
     try
     {
             int tmp= 0;
             FileWriter filewriter = new FileWriter(file);
             while((tmp=System.in.read()) != -1)
     {
     filewriter.write((char) tmp);
     }
     filewriter.close();
     }
     catch (IOException e)
     {
     System.out.println(" IO erro" + e);
     }
     System.out.println(" File.length()="+file.length());
    }
   }
```

Output:

```
C:\Java\Block3Unit3> Java TextFileOutput_Appl
Hi, test input start here.
^Z
```

```
File.length()=28
C:\Java\Block3Unit3>
```

4.5 STREAM TOKENIZER

Stream Tokenizer is introduced in JDK 1.1 to improve the wc() [Word Count] method, and to provide a better way for pattern recognition in input stream. (do you know how wc() mehod is improved by Stream Tokenizer ?) during writing program. Many times when reading an input stream of characters, we need to parse it to see if what we are reading is a word or a number or something else. This kind of parsing process is called "tokenizing". A "token" is a sequence of characters that represents some abstract values stream. Tokenizer can identify members, quoted string and many other comment styles.

Stream Tokenizer Class is used to break any input stream into tokens, which are sets of bits delimited by different separator. A few separators already added while you can add other separators. It has two contributors given below:

There are 3 instance variable as defined in streamtokenizer **nval**,**sval** and **ttype.** nval is public double used to hold number, sval is public string holds word, and ttype is pubic integer to indicates the type of token. nextToken() method is used with stream tokenizer to parse the next token from the given input stream. It returns the value of token type **ttype**, if the token is word then ttype is equal to TT_WORD. There are some other values of ttype also which you can see in Table 1 given below. You can check in the given program below that we are using TT_EOF in the same way as we have used end of file (EOF) in our previous example. Now let me tell you how wc() is improved, different token types are available stream tokenizer which we can use to parse according to our choice, which were limitation of wc() method also.

Table 1: Token Types

Token Types	Meaning
TT_WORD	String word was scanned (The string field **sval** contains the word scanned)
TT_NUMBER	A number was scanned (only decimal floating point number)
TT_EOL	End – of - line scanned
TT_EOF	End – of –file – scanned

```
// Program Start here
import java.io.*;
public class tokenizer
{
  public static void main(String args[])
  {
     String sample="this 123 is an 3.14 \n simple test";
      try
      {
      InputStream  in;
      in = new StringBufferInputStream(sample);
      StreamTokenizer parser = new StreamTokenizer(in);
      while(parser.nextToken() != StreamTokenizer.TT_EOF)
      {
       if (parser.ttype == StreamTokenizer.TT_WORD)
          System.out.println("A word"+parser.sval);
      else if (parser.ttype == StreamTokenizer.TT_NUMBER)
          System.out.println("A number: "+parser.nval);
```

```
            else if (parser.ttype == StreamTokenizer.TT_EOL)
            System.out.println("EOL");
            else if (parser.ttype == StreamTokenizer.TT_EOF)
            throw new IOException("End of FIle");
            else
            System.out.println("other" +(char) parser.ttype);
           }
           }
           catch (IOException e)
           {
           System.out.println(" IO erro" + e);
           }
       }
}
```

Output:

```
A word: this
A number: 123
A word: is
A word: an
A number: 3.14
A word: simple
A word: test
```

After passing the input stream, each token is identified by the program and the value of token is given in the output of the above program.

4.6 SERIALIZATION

You can read and write text, but with Java you can also read and write objects to files. You can say object I/O is serialization where you read/write state of an object to a byte stream. It plays a very important role sometimes when you are interested in saving the state of objects in your program to a persistent storage area like file. So that further you can De-Serialize to restore these objects, whenever needed. I have a question for you, why is implementing of serialization very difficult with other language and it is easy with Java?

In other languages it may be difficult, since objects may contain references to another objects and it may lead to circular references. Java efficiently implements serialization, though we have to follow different Conditions; we can call them, Conditions for serializability.

If you want to make an object serialized, the class of the object must be declared as public, and must implement serialization. The class also must have a no argument constructer and all fields of class must be serializable.

Java gives you the facility of serialization and deserialization to save the state of objects. There are some more functions which we often used on files like compression and encryption techniques. We can use these techniques on files using Externalizable interface. For more information you can refer to books given in references.

☞ **Check Your Progress 2**

1) Why is PushbackInputStream is used?

……………………………………………………………………………………

……………………………………………………………………………………

2) Write a program to print all primitive data values into a File using FileWriter stream wrapped with a Printwriter.

………………………………………………………………………………………

………………………………………………………………………………………

3) How would you append data to the end of a file? Show the constructor for the class you would use and explain your answer. .

………………………………………………………………………………………

……………………………………………………………………………………

4.7 BUFFERED STREAM

What is Buffer? Buffer is a temporary storage place where the data can be kept before it is needed by the program that reads or writes data. By using buffer, you can get data without always going back to the original source of data. In JAVA, these Buffers allow you to do input/output operations on more than a byte at a time. It increases the performance, and we can easily manipulate, skip, mark or reset the stream of byte also.

Buffered Stream classes manipulate sequenced streams of byte data, typically associated with binary format files. For example, binary image file is not intended to be processed as text and so conversion would be appropriate when such a file is read for display in a program. There are two classes for bufferstream BufferInputStream and BufferedOutputStream.

A *Buffered input stream* fills a buffer with data that hasn't been handled yet. When a program needs this data, it looks to the buffer first before going to the original stream source. A *Buffered input stream* is created using one of the two constructors:

BufferedInputStream (Input_Stream) – Creates a buffered input stream for the specified Input Stream object.

BufferedInputStream (Input_Stream, int) – Creates a specified buffered input stream with a buffer of int size for a specified Inputstream object.

The simplest way to read data from a buffered input stream is to call its read () method with no argument, which returns an integer from 0 to 255 representing the next byte in the stream. If the end of stream is reached and no bytes are available, –1 is returned.
A *Buffered output stream* is created using one of the two constructors:

BufferedOutputStream(OutputStream) – Creates a buffered output stream for the specified Output Stream object.

BufferedOutputStream (OutputStream, int) – Creates a buffered output stream with a buffer of int size for the specified outputstream object.

The output stream's write (int) method can be used to send a single byte to the stream. write (byte [], int, int) method writes multiple bytes from the specified byte array. It specifies the starting point, and the number of bytes to write. When data is directed to a buffered stream, it is not output to its destination until the stream fills or buffered stream *flush()* method is used.

In the following example, we have explained a program for you, where we are creating buffer input stream from a file and writing with buffer output stream to the file.

```
// program
import java.lang.System;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.SequenceInputStream;
import java.io.IOException;
public class BufferedIOApp
{
public static void main(String args[]) throws IOException
{
SequenceInputStream f3;
FileInputStream f1 = new FileInputStream("String_Test.Java");
FileInputStream f2 = new FileInputStream("textOutput.txt");
f3 = new SequenceInputStream(f1,f2);
BufferedInputStream inStream = new BufferedInputStream(f3);
BufferedOutputStream outStream = new BufferedOutputStream(System.out);
inStream.skip(500);
boolean eof = false;
int byteCount = 0;
while (!eof)
{
int c = inStream.read();
if(c == -1) eof = true;
else
{
outStream.write((char) c);
++byteCount;
}
}
String bytesRead = String.valueOf(byteCount);
bytesRead+=" bytes were read\n";
outStream.write(bytesRead.getBytes(),0,bytesRead.length());
System.out.println(bytesRead);
inStream.close();
outStream.close();
f1.close();
f2.close();
}
}
```

Character Streams

Character streams are used to work with any text files that are represented by ASCII characters set or Unicode set. Reading and writing file involves interacting with peripheral devices like keyboard printer that are part of system environment. As you know such interactions are relatively slow compared to the speed at which CPU can operate. For this reason *Buffered reader and Buffered Writer* classes are used to overcome from difficulties posed by speed mismatch.

When a read method of a *Buffered Reader* object, is invoked it might actually read more data from the file then was specifically asked for. It will hold on to additional data read from the file that it can respond to the next read request promptly without making an external read from the file.
Similarly *Buffered Writer* object might bundle up several small write requests in its own internal buffer and only make a single external write to file operation when it considers that it has enough data to make the external write when its internal buffer is full, or a specific request is made via call to its *flush* method. The external write occurs.

It is not possible to open a file directly by constructing a Buffered Reader or Writer Object. We construct it from an existing FileReader or FileWriter object. It has an advantage of not duplicating the file- opening functionality in many different classes

The following constructors can be used to create a BufferedReader:
BufferedReader (Reader) – Creates a buffered character stream associated with the specified Reader object, such as FileReader.

BufferedReader (Reader, int) – Creates a buffered character stream associated with the specified Reader object, and with a buffered of int size.

Buffered character string can be read using read() and read(char[],int,int). You can read a line of text using readLine() method. The readLine() method returns a String object containing the next line of text on the stream, not including the characters or characters that represent the end of line. If the end of stream is reached the value of string returned will be equal to **null.**

In the following program we have explained the use of BufferedReader where we can calculate statistic (No. of lines, No. of characters) of a file stored in the hard disk named “text.txt”.

```
// Program
import Java.io.*;
public class FileTest
{
public static void main(String[] args) throws IOException
{
String s = "textOutput.txt";
File f = new File(s);
if (!f.exists())
{
System.out.println("\'" + s + "\' does not exit. Bye!");
return;
}
// open disk file for input
BufferedReader inputFile = new BufferedReader(new FileReader(s));
// read lines from the disk file, compute stats
String line;
int nLines = 0;
int nCharacters = 0;
while ((line = inputFile.readLine()) != null)
{
nLines++;
nCharacters += line.length();
}
// output file statistics
System.out.println("File statistics for \'" + s + "\'...");
System.out.println("Number of lines = " + nLines);
System.out.println("Number of characters = " + nCharacters);
inputFile.close();
}
}
```

Output:

```
File statistics for 'textOutput.txt'...
Number of lines = 1
Number of characters = 26
```

4.8 PRINT STREAM

A *PrintStream* is a grandchild of OutputStream in the Java class hierarchy–it has methods implemented that print lines of text at a time as opposed to each character at a time. It is used for producing formatted output.

The original intent of *PrintStream* was to print all of the primitive data types and String objects in a viewable format. This is different from DataOutputStream, whose goal is to put data elements on a stream in a way that DataInputStream can portably reconstruct them.

Constructors

PrintStream (OutputStream out): PrintStream(OutputStream out) creates a new print stream. This stream will not flush automatically; returns a reference to the new PrintStream object.

PrintStream(OutputStream out, boolean autoFlush): Creates a new print stream. If autoFlush is true, the output buffer is flushed whenever (a) a byte array is written, (b) one of the println() methods is invoked, or (c) a newline character or byte ('\n') is written; returns a reference to the new PrintStream object.

Methods

flush(): flushes the stream; returns a void

print(char c): prints a character; returns a void

println(): terminates the current line by writing the line separator string; returns a void

println(char c): prints a character and then terminates the line; returns a void

Two constructors of PrintStream are deprecated by java 1.1 because PrintStream does not handle Unicode character and is thus not conveniently internationalized. Deprecating the constructors rather than the entire class allows existing use of System.out and Sytem.err to be compiled without generating deprecation warnings. Programmers writing code that explicitly constructs a PrintStream object will see deprecating warning when the code is compiled. Code that produces textual output should use the new PrintWriter Class, which allows the character encoding to be specified or default encoding to be accepted. System.out and System.err are the *only* PrintStream object you should use in programs.

PrintWriter

To open a file output stream to which text can be written, we use the PrintWriter class. As always, it is best to buffer the output. The following sets up a buffered file writer stream named outFile to write text into a file named save.txt. The Java statement that will construct the required PrintWriter object and its parts is:

PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter("save.txt"));

The object outFile, above, is of the PrintWriter class, just like System.out. If a string, s, contains some text, it is written to the file as follows:

outFile.println(s);

When finished, the file is closed as expected:

outFile.close();

In the following program we are reading text from the keyboard and writing the text to a file called junk.txt.

```
//program
import Java.io.*;
class FileWrite
{
public static void main(String[] args) throws IOException
{
        BufferedReader stdin = new BufferedReader(new
        InputStreamReader(System.in));
        String s = "junk.txt";
        File f = new File(s);
        if (f.exists())
        {
        System.out.print("Overwrite " + s + " (y/n)? ");
        if(!stdin.readLine().toLowerCase().equals("y"))
        return;
        }
        // open file for output
        PrintWriter outFile = new PrintWriter(new BufferedWriter(new
        FileWriter(s)));
        System.out.println("Enter some text on the keyboard...");
        System.out.println("(^z to terminate)");
        String s2;
        while ((s2 = stdin.readLine()) != null)
        outFile.println(s2);
        outFile.close();
  }
  }
```

Output:

```
Enter some text on the keyboard...
(^z to terminate)
How are you? ^z
```

4.9 RANDOM ACCESS FILE

The character and byte stream are all sequential access streams whose contents must be read or written sequentially. In contrast, Random Access File lets you randomly access the contents of a file. The *Random Access File* allows files to be accessed at a specific point in the file. They can be opened in read/write mode, which allows updating of a current file.

The *Random Access File* class is not derived from InputStream and OutputStream. Instead it implements DataInput and DataOutput and thus provides a set of methods from both DataInputStream and DataOutputStream.

An object of this class should be created when access to binary data is required but in non-sequential form. The same object can be used to both read and write the file.

To create a new Random Access file pass the name of file and mode to the constructor. The mode is either “r” (read only) or “rw” (read and write)
An *IllegalArgumentException* will be thrown if this argument contains anything else.

Following are the two constructors that are used to create RandomAccessFile:

RandomAccessFile(String s, String mode) throws IOException and s is the file name and mode is “r” or “rw”.
RandomAccessFile(File f, String mode) throws IOException and f is an File object, mode is ‘r’ or ‘rw’.

Methods

long length() : returns length of bytes in a file
void seek(long offset) throws IOException: position the file pointer at a particular point in a file. The new position is current position plus the offset. The offset may be positive or negative.
long getFilePointer() throws IOException : returns the index of the current read write position within the file.
void close() throws IOException: close fine and free the resource to system.

To explain how to work with random access files lets write a program, here in the following program we are writing a program to append a string “I was here” to all the files passed as input to program.

```
import Java.io.*;
public class AppendToAFile
{
  public static void main(String args[])
  {
   try
    {
    RandomAccessFile raf = new RandomAccessFile("out.txt","rw");
   // Position yourself at the end of the file
    raf.seek(raf.length());
   // Write the string into the file.Note that you must explicitly handle line breaks
    raf.writeBytes("\n I was here\n");
   }
   catch (IOException e)
   {
    System.out.println("Eror Opening a file" + e);
   }
  }
  }
```

After executing this program: “I was here” will be appended in out.txt file.

☞ Check Your Progress 3

1) Write a program to read text from the keyboard and write the text to a file called junk.txt.

…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………

2) Create a file test.txt and open it in read write mode.Add 4 lines to it and randomly read from line 2 to 4 and line 1.

…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………

3) Write a program using FileReader and FileWriter, which copy the file f1.text one character at a time to file f2.text.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

4.10 SUMMARY

This unit covered the Java.io package. Where you start with two major stream of Java I/O, the InputStream and OutputStream, then learnt the standard methods available in each of these classes. Next, you studied other classes that are in the hierarchy derived from these two major classes and their method to allow easy implementation of different stream types in Java.

In the beginning of this unit we studied File class and its methods for handling files and directories. Further we have looked at the different classes of the Java.io package. We have worked with streams in two directions: pulling data over an input stream or sending data from a program using an output stream. We have used byte stream for many type of non-textual data and character streams to handle text. Filters were associated with streams to alter the way information was delivered through the stream, or to alter the information itself.

Let us recall all the uses and need of the classes we just studied in this unit. Classes based upon the InputStream class allow byte-based data to be input and similarly OutputStream class allows the output. The DataInputStream and DataOutputStream class define methods to read and write primititve datatype values and strings as binary data in host independent format. To read and write functionality with streams Inputstreamreader and outputstreamwriter class is available in Java.io package. The RandomAccessFile class that encapsulates a random access disk file allows a file to be both read and written by a single object reference. Reading functionality for system.in can be provided by using input stream reader. Writing functionality for system.out can be provided by using Print writing. The stream tokenizer class simplifies the parsing of input files which consist of programming language like tokens.

4.11 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Unicode is a 16-bit code having a large range in comparison to previous ASCII code, which is only 7 bits or 8 bits in size. Unicode can represent all the characters of all human languages. Since Java is developed for designing Internet applications, and worldwide people can write programs in Java, transformation of one language to another is simple and efficient. Use of Unicode also supports platform independence in Java.

2)

```
      //program
      import Java.io.*;
      class rename
{
public static void main(String args[])
{
File f1 = new File("out1.txt");
File f2 = new File("out2.txt");
if(f1.exists())
{
System.out.println(f1 + " File exists");
System.out.println("size of file is: " + f1.length() + "bytes" );
f1.renameTo(f2);
System.out.println("Rename file is: " + f2);
System.out.println("deleting file : " + f1);
f1.delete();
}
else
System.out.println("f1" + "file is not existing");
}
}
```

Output:

```
out1.txt File exists
size of file is: 22bytes
Rename file is: out2.txt
deleting file : out1.txt
```

3)

```
//program
import Java.io.*;
class ReadWrite
{
 public static void main(String args[] )throws IOException
 {
 byte[] c= new byte[10];
 System.out.println("Enter a string of 10 character");
 for (int i=0;i < 10;i++)
 c[i]=(byte)System.in.read();
 System.out.println("Following is the string which you typed: ");
 System.out.write(c);
 }
}
```

Output:

```
C:\Java\Block3Unit3>Java ReadWrite
Enter a string of 10 character
Hi how is life
Following is the string which you typed:
Hi how is
```

Check Your Progress 2

1) PushbackInputStream is an input stream that can unread bytes. This is a subclass of FilterInputStream which provides the ability to unread data from a stream. It maintains a buffer of unread data that is supplied to the next read operation. It is used in situations where we need to read some unknown number of data bytes that are delimited by some specific byte value. After reading the specific byte program can perform some specific task or it can terminate.

2) This program use FileWriter stream wrapped with a Printwriter to write binary data into file.

```
        //program
        import Java.io.*;
public class PrimitivesToFile_Appl
{
 public static void main(String args[])
 {
        try
        {
        FileWriter filewriter = new FileWriter("prim.txt");
        // Wrap the PrintWriter with a PrintWriter
        // for sending the output t the file
        PrintWriter printWriter = new PrintWriter(filewriter);
        boolean b=true;
        byte    by=127;
        short   s=1111;
        int     i=1234567;
        long    l=987654321;
        float   f=432.5f;
        double  d=1.23e-15;
        printWriter.print(b);
        printWriter.print(by);
        printWriter.print(s);
        printWriter.print(i);
        printWriter.print(l);
        printWriter.print(f);
        printWriter.print(d);
        printWriter.close();
        }
        catch (Exception e)
        {
        System.out.println("IO erro" + e);
    }
   }
 }
```

Output:
File prim.txt will contain : true12711111234567987654321432.51.23E-15

3) First let us use the Filewriter and BufferedWriter classes to append data to the end of the text file. Here is the FileWriter constructor, you pass in true value in second argument to write to the file in append mode:

```
FileWriter writer = new FileWriter (String filename, boolean append);
```

An alternate answer is to use RandomAccessFile and skip to the end of the file and start writing

```
RandomAccessFile file = new  RandomAccessFile(datafile,"rw");
file.skipBytes((int)file.length()); //skip to the end of the file
file.writeBytes("Add this text to the end of datafile"); //write at the end the of the file
file.close();
```

Check Your Progress 3

1)

```
        //program
        import Java.io.*;
        class FileWriteDemo
        {
        public static void main(String[] args) throws IOException
        {
        // open keyboard for input

BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
//output file 'junk.txt'
String s = "junk.txt";
File f = new File(s);
if (f.exists())
{
System.out.print("Overwrite " + s + " (y/n)? ");
if(!stdin.readLine().toLowerCase().equals("y"))
return;
}
PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter(s)));
System.out.println("Enter some text on the keyboard...");
System.out.println("(^z to terminate)");
String s2;
while ((s2 = stdin.readLine()) != null)
outFile.println(s2);
outFile.close();
}
}
```

2)

```
//program
import Java.lang.System;
import Java.io.RandomAccessFile;
import Java.io.IOException;
public class RandomIOApp
{
 public static void main(String args[]) throws IOException
{
RandomAccessFile file = new RandomAccessFile("test.txt","rw");
file.writeBoolean(true);
file.writeInt(123456);
file.writeChar('j');
file.writeDouble(1234.56);
file.seek(1);
System.out.println(file.readInt());
System.out.println(file.readChar());
System.out.println(file.readDouble());
file.seek(0);
System.out.println(file.readBoolean());
file.close();
}
}
Output:
123456
j
1234.56
true
```

3)

```
        //program
        import Java.io.*;
        public class CopyFile
        {
        public static void main(String[] args) throws IOException,
        FileNotFoundException
        {
        File inFile = inFile = new File("out2.txt");
        FileReader in = new FileReader(inFile);
        File outFile = new File("f2.txt");
FileWriter out = new FileWriter(outFile);
// Copy the file one character at a time.
int c;
while ((c = in.read()) != -1)
out.write(c);
in.close();
out.close();
System.out.println("The copy was successful.");
}
}
```

Output:

```
The copy was successful.
```