Thread:

A thread represents a separate path of execution of group of statements. In a java program, if we write a group of statements, then these statements are executed by JVM on by one. This execution is called a thread, because JVM uses a thread to execute these statements. This means that in every java program, there is always a thread running internally. This thread used by JVM to execute the program statements.

Program: write a java program to find the thread used by JVM to execute the statements.

```
class Demo
{
	public static void main(String[] args)
	{
		System.out.println("Welcome");
		Thread t=Thread.currentThread();
		System.out.println("Current Thread= "+t);
		System.out.println("Thread Name= "+t.getName());
	}
}
```

Output: Welcome

Current Thread= Thread[main, 5, main]

Thread Name= main

In the above program, `currentThread()` is a static method in Thread class. So we called it as `Thread.currentThread()`. It displayed as Thread[main, 5, main]. Here, the first main indicated the name of the thread running the current code. We get 5 which is a number representing the priority numbers will range from 1 to 10. 1 is minimum priority, and 10 is the maximum priority of a thread. A thread is group of statements, and these statements are executed in two ways.

1. Single tasking
2. Multi tasking

1. Single tasking:

In single tasking only one task is given to the processor at a time. This means we are waiting lot of time and micro processor has to sit idle with doing any job for long time.

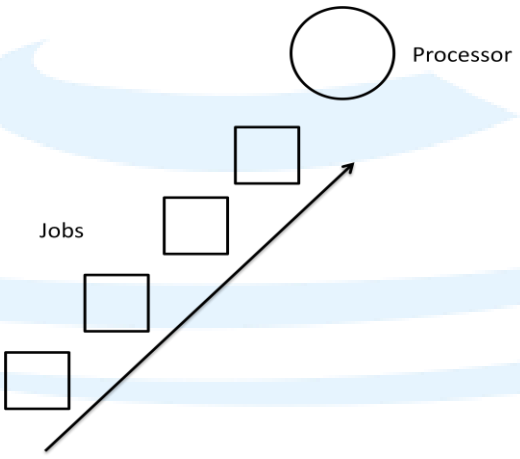


Fig: Single Tasking

2. Multi Tasking:

In multi tasking, several jobs are executed at a time. These jobs are executed by the processor simultaneously. One processor has executed the several tasks by using round robin method.

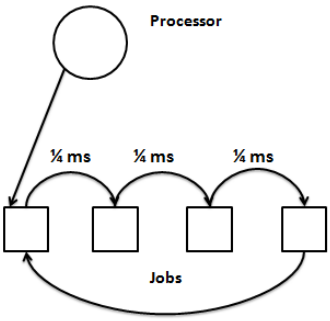
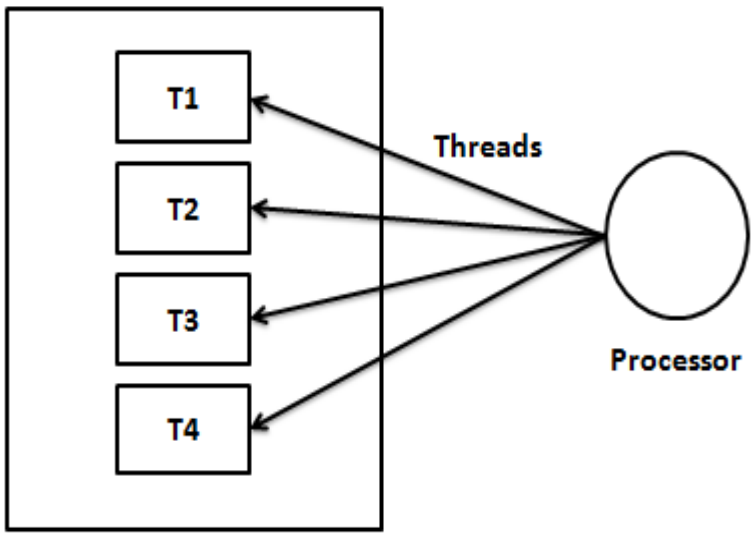


Fig: Process based Multi tasking

In multi tasking there are two types:

a) Process based Multi tasking
b) Thread based Multi tasking

- In process based multi tasking several programs are executed at a time by the microprocessor.
- In Thread Based Multi tasking several parts of programs are executed by the processor.



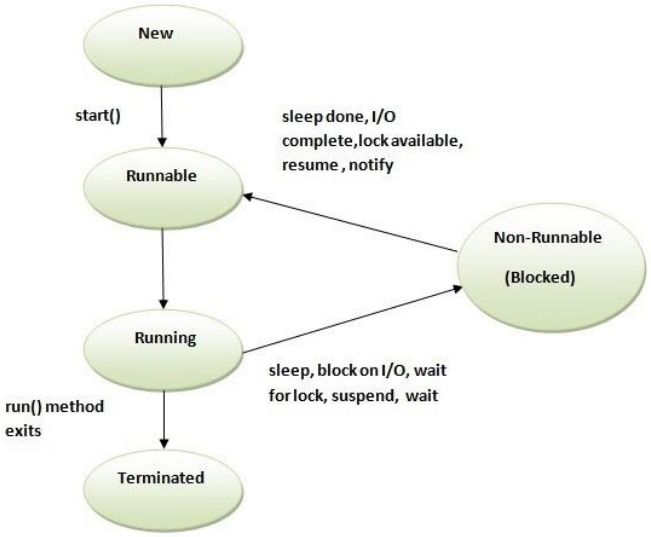
Note: Threads are light weight process because they utilize minimum resources of the system. This means they take less memory and less processor time.

Uses of Thread:

- Threads are mainly used in server-side programs to serve the needs of multiple clients on network or internet.
- Threads are also used to create games and animation. Animation means moving of objects from one place to another.

Thread Life Cycle:

1. **New:** The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. **Runnable:** The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. **Running:** The thread is in running state if the thread scheduler has selected it.
4. **Non-Runnable (Blocked):** This is the state when the thread is still alive, but is currently not eligible to run.
5. **Terminated:** A thread is in terminated or dead state when its run() method exits.



Creating Thread and Running it:

To create a thread we have to follow these steps:

- Create a class that extends Thread class or implements Runnable interface. Both the Thread class and Runnable interface are found in java.lang package.

```
class MyThread extends Thread
or, class MyThread implements Runnable
```

- Now in this class we have write a run() method as:

```
public void run()
{
}
```

  By default, this `run()` method is recognized and executed by a thread.

- Create an object to MyThread, so that the `run()` method is available for execution.

```
MyThread obj = new MyThread();
```

- Now, create a thread and attach the thread to the object obj.

```
Thread t = new Thread(obj);
Or, Thread t = new Thread(obj, "Thread Name");
```

- Run the thread.

```
t.start();
```

  This start method executes `run()` method in MyThread class.

```
import java.lang.*;
class MyThread extends Thread
{
   public void run()
   {
         For(int i=0;i<10;i++)
         System.out.println(i);
   }
   public static void main(String[] args)
   {
         MyThread mt=new MyThread();
         Thread t=new Thread(mt);
         t.start();
   }
}
```

Terminating a Thread:

A thread will terminate automatically when it comes out or run() method. To terminate the thread on our own, we have to design our own logic.

- Create a Boolean variable stop and initialize it false.

  ```
  boolean stop = false;
  ```

- Let us assume that we want to terminate the thread when the user process <Enter> key. So, when the user presses that button, make the Boolean type variable as true.

  ```
  stop = true;
  ```

- check this variable in run() method and when it is true, make the thread return from the run() method.

  ```
  public void run()
  {
        if(stop == true) return;
  }
  ```

Program:

```
import java.lang.*;
class MyThread extends Thread
{
      boolean stop=false;
      public void run()
      {
            for(int i=0;i<1000;i++)
            {
                  if(stop==true) return;
                  System.out.println(i);
            }
      }
      public static void main(String[] args)
      {
```

```
            MyThread mt=new MyThread();
            Thread t=new Thread(mt);
            t.start();
            system.in.read();
            mt.stop=true;
      }
}
```

Sleep method in Threads:

- `sleep()` method in Threads is used to stop the execution of a thread for specified time. This time given as parameter to the sleep method in milliseconds. The Syntax is:

```
Thread.sleep(int milliseconds);
```

- This method can raise an interruptedException so the programmer should handle the exception.

Single tasking using Thread:

In single tasking tasks are executed one by one.

Program:

```
import java.lang.*;
class MyThread extends Thread
{
      public void run()
      {
            task1();
            task3();
            task2();
      }
      public void task1()
      {
            System.out.println("Task 1");
      }
      public void task2()
      {
            System.out.println("Task 2");
      }
      public void task3()
      {
            System.out.println("Task 3");
      }
      public static void main(String[] args)
      {
            MyThread mt=new MyThread();
            Thread t=new Thread(mt);
            t.start();
      }
}
```

Multi tasking using Thread:

In multi tasking several threads are executed at a time. For this purpose we want two or more threads. For example, to perform 2 tasks, we can take 2 threads and attach them to the 2 tasks. Then those tasks are simultaneously executed by the two threads.

Program:

```
import java.lang.*;
class MyThread extends Thread
{
	String name="";
	MyThread(String s)
	{
		name=s;
	}
	public void run()
	{
		for(int i=0;i<20;i++)
		{
			System.out.println(name+" "+i);
		}
	}
	public static void main(String[] args)
	{
		MyThread mt1=new MyThread("Hello");
		MyThread mt2=new MyThread("Welcome");
		Thread t1=new Thread(mt1);
		Thread t2=new Thread(mt2);
		t1.start();
		t2.start();
	}
}
```

Multiple threads acting on single Object:

We can create two or more threads and pass single object to those threads. But in this case sometimes we can some unreliable results.

Program:

```
class Reserve extends Thread
{
	int avail=1;
	int wanted;
	Reserve(int i)
	{
		wanted=i;
	}
	public void run()
	{
		System.out.println("Available Berths= "+avail);
```

```
            if(avail>=wanted)
            {
                  System.out.println(wanted+" Seat(s) Booked ");
                  try{
                        Thread.sleep(1000);
                  }catch(Exception e){}
                  avail=avail-wanted;
            }
            else
            {
                  System.out.println("Sorry, No Berths");
            }
      }
}
class Unsafe
{
      public static void main(String[] args)
      {
            Reserve r=new Reserve(1);
            Thread t1=new Thread(r);
            Thread t2=new Thread(r);
            t1.start();
            t2.start();
      }
}
```

Output:

```
Available Berths= 1
Available Berths= 1
1 Seat(s) Booked
1 Seat(s) Booked
```

Please observe the output in the preceding program. It is absurd. It has allotted the same berth for both passengers. In the above first thread t1 enters into try block and it will sleep for 1 second for printing a ticket. When the first thread is sleeping the second thread will enters into run() method, it also sees that there is 1 berth remaining. The reason is available seats are not updated by the first thread. So, the second thread also sees 1 berth is available, and it allots the same berth for second person.

Since, both threads are acting on same object simultaneously, the result is unreliable. The solution for this problem is ***Thread Synchronization***.

Thread Synchronization:

When a thread is acting on an object, preventing any other thread from acting on the same object is called ‘Thread Synchronization’ or ‘Thread Safe’. The object on which the threads are synchronized is called ‘synchronized object’. Thread synchronization is recommended when multiple threads are used on the same object.

Synchronized object is like a locked object, locked on thread. It is like a room with only one door. A person has entered the room and locked from it from behind. The second person will wait till the first person comes out. So, this object is called ‘mutex’.

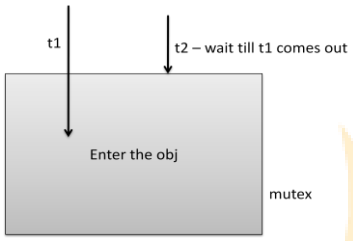


Fig: Thread Synchronization

How can we synchronize the object? There are two types of doing this.

- Using **synchronized block:** Here, we can embed a group of statements of objects inside the run method within a synchronized block, as shown here:

```
synchronized(this)
{
        Statements;
}
```

- Using **synchronized keyword:** we can synchronize an entire method by using synchronized keyword.

```
synchronized void display()
{
        Statements;
}
```

Program:

```
class Reserve extends Thread
{
       int avail=1;
       int wanted;
       Reserve(int i)
       {
              wanted=i;
       }
       public void run()
       {
              synchronized(this)
              {
              System.out.println("Available Berths= "+avail);
              if(avail>=wanted)
              {
                     System.out.println(wanted+" Seat(s) Booked ");
                     try{
                            Thread.sleep(1000);
                     }catch(Exception e){}
                     avail=avail-wanted;
              }
              else
              {
                     System.out.println("Sorry, No Berths");
              }
              }
       }
}
class Safe
```

```
{
	public static void main(String[] args)
	{
		Reserve r=new Reserve(1);
		Thread t1=new Thread(r);
		Thread t2=new Thread(r);
		t1.start();
		t2.start();
	}
}
```

Output:

Available Berths= 1
1 Seat(s) Booked
Available Berths= 0
Sorry, No Berths

Thread class methods:

- To create a thread, we can use the following forms:

```
Thread t1 = new Thread();
Thread t2 = new Thread(obj);
Thread t3 = new Thread(obj, “Thread Name” );
```

- To know currently running thread:

```
Thread t = Thread.curentThread();
```

- To start a thread:

```
t.start();
```

- To stop execution of a thread for specified time:

```
Thread.sleep(int milliseconds);
```

- To get the name of thread:

```
String name = t.getName();
```

- To set new name to the thread:

```
t.setName(“New Name”);
```

- To get the priority of a thread:

```
int pno = t.getPriority();
```

- To set priority to the thread:

```
t.setPriority(int pno);
```

- To test if a thread is still alive:

```
t.isAlive();
```

- To wait till a thread dies:

```
t.join();
```

Deadlock of Threads:

Even if we synchronize the threads, there is possibility of other problems like ‘deadlock’. Daily, thousands of people book tickets in trains and cancel tickets also. If a programmer is to develop code for this, he may visualize the booking tickets and cancelling them are reverse procedures. Hence, he will write these 2 tasks as separate and opposite tasks, and assign 2 different threads to do these tasks simultaneously.

Program:

```
class BookTicket extends Thread
{
	Object train, comp;
	BookTicket(Object train, Object comp)
	{
		this.train=train;
		this.comp=comp;
	}
	public void run()
	{
		synchronized(train)
		{
			System.out.println("BookTicket locked on train");
			try{
				Thread.sleep(150);
			}catch(Exception e){}
			System.out.println("BookTicket waiting to lock on
						Compartment");
			synchronized(comp)
			{
				System.out.println("BookTicket locked on
							Compartment");
			}
		}
	}
}
class CancelTicket extends Thread
{
	Object train, comp;
	CancelTicket(Object train, Object comp)
	{
		this.train=train;
		this.comp=comp;
	}
	public void run()
	{
		synchronized(comp)
		{
			System.out.println("CancelTicket locked on
						Compartment");
			try{
				Thread.sleep(200);
			}catch(Exception e){}
			System.out.println("CancelTicket waiting to lock on
						Train");
			synchronized(train)
```

```
                {
                        System.out.println("CancelTicket locked on
                                                Train");
                }
        }
    }
}
class Deadlock
{
    public static void main(String[] args)
    {
        Object train = new Object();
        Object comp = new Object();
        BookTicket bt=new BookTicket(train,comp);
        CancelTicket ct=new CancelTicket(train,comp);
        Thread t1=new Thread(bt);
        Thread t2=new Thread(ct);
        t1.start();
        t2.start();
    }
}
```

Output:

BookTicket locked on train
CancelTicket locked on Compartment
BookTicket waiting to lock on Compartment
CancelTicket waiting to lock on Train
.....

Please observer above output, the program is not terminated, BookTicket thread is waiting for Compartment object and CancelTicket thread is waiting for Train object. This waiting continues forever.

Avoiding Deadlocks in a program:

There is no specific solution for the problem of deadlocks. It depends on the logic used by the programmer. The programmer should design his program in such a way, that it does not form any deadlocks. For example, in the preceding program, if the programmer used the threads in such a way that the CancelTicket thread follows the BookTicket, then he could have avoided the deadlock situation.

Program:

```
class BookTicket extends Thread
{
    Object train, comp;
    BookTicket(Object train, Object comp)
    {
        this.train=train;
        this.comp=comp;
    }
    public void run()
    {
        synchronized(train)
        {
```

```
                    System.out.println("BookTicket locked on train");
                    try{
                          Thread.sleep(150);
                    }catch(Exception e){}
                    System.out.println("BookTicket waiting to lock on
                                        Compartment");
                    synchronized(comp)
                    {
                          System.out.println("BookTicket locked on Compartment");
                    }
             }
      }
}
class CancelTicket extends Thread
{
      Object train, comp;
      CancelTicket(Object train, Object comp)
      {
             this.train=train;
             this.comp=comp;
      }
      public void run()
      {
             synchronized(train)
             {
                    System.out.println("CancelTicket locked on Train");
                    try{
                          Thread.sleep(200);
                    }catch(Exception e){}
                    System.out.println("CancelTicket waiting to lock on Compartment");
                    synchronized(comp)
                    {
                          System.out.println("CancelTicket locked on Compartment");
                    }
             }
      }
}
class Deadlock
{
      public static void main(String[] args)
      {
             Object train = new Object();
             Object comp = new Object();
             BookTicket bt=new BookTicket(train,comp);
             CancelTicket ct=new CancelTicket(train,comp);
             Thread t1=new Thread(bt);
             Thread t2=new Thread(ct);
             t1.start();
             t2.start();
      }
}
```

Output:

BookTicket locked on train
BookTicket waiting to lock on Compartment
BookTicket locked on Compartment
CancelTicket locked on Train
CancelTicket waiting to lock on Compartment
CancelTicket locked on Compartment

Thread Communication:

In some cases, two or more threads should communicate with each other. For example, a Consumer thread is waiting for Producer to produce the data. When the Producer thread completes the production of data, then the Consumer thread should take that data and use it.

In the Producer class, we take StringBuffer object to store data; in this case, we take numbers from 1 to 10. These numbers are added to StringBuffer object. We take another Boolean variable status, and initialize it to false. This idea is to make this status is true when production of numbers is completed. Producing data is done by appending numbers into StringBuffer using for loop. This may take some time.

In the Consumer class, it will check for status is true or not. If status is true, the Consumer takes the data from StringBuffer and uses it. If status is false, then Consumer will sleep for some time and then checks status.

Program:

```
import java.util.*;
class Producer extends Thread
{
	StringBuffer sb=new StringBuffer();
	boolean status=false;
	public void run()
	{
		synchronized(sb)
		{
			for(int i=1;i<=10;i++)
			{
				sb.append(i+" : ");
				System.out.println("Appending");
				try{
					Thread.sleep(100);
				}
				catch(Exception ie) { }
			}
			status=true;
		}
	}
}
class Consumer extends Thread
{
	Producer prod;
	Consumer(Producer prod)
	{
		this.prod=prod;
	}
	public void run()
	{
		synchronized(prod.sb)
		{
			try{
				while(prod.status==false)
```

```
                                Thread.sleep(50);
                        }
                        catch(Exception e) { }
                        System.out.println("Data is: "+prod.sb);
                }
        }
}
class Communciate
{
        public static void main(String[] args)
        {
                Producer p=new Producer();
                Consumer c=new Consumer(p);
                Thread t1=new Thread(p);
                Thread t2=new Thread(c);
                t1.start(); //Consumer thread will start first
                t2.start();
        }
}
```

Output:

```
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Data is: 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 :
```

In this way, the producer and consumer can communicate with each other. But this NOT efficient way of communication. Why? Consumer checks the status at some point of time and finds it false. So, it goes into sleep for next 10 milliseconds. Meanwhile, the data provider may be over. But Consumer comes out of sleep after 10 milliseconds and then only it can check status is true. This means that there may be 1 to 9 milliseconds to receive the data after its actual production is completed.

To improve the efficiency of communication between threads. java.lang.Object class provides 3 methods for this purpose.

- **`obj.notify()`:** This method releases an object and sends a notification to waiting thread that the object is available.
- **`Obj.notifyAll()`:** This method is useful to send notification to all waiting threads at once that the object is available.
- **`Obj.wait()`:** this method makes a thread wait for the object till it receives a notification from a notify() method or notifyAll() method.

Program:

```
import java.util.*;
class Producer extends Thread
{
	StringBuffer sb=new StringBuffer();
	public void run()
	{
		synchronized(sb)
		{
			for(int i=1;i<=10;i++)
			{
				sb.append(i+" : ");
				System.out.println("Appending");
				try{ Thread.sleep(100); }
				catch(InterruptedException ie) { }
			}
			sb.notify();
		}
	}
}
class Consumer extends Thread
{
	Producer prod;
	Consumer(Producer prod)
	{
		this.prod=prod;
	}
	public void run()
	{
		synchronized(prod.sb)
		{
			try{
				prod.sb.wait();
			}
			catch(Exception e) { }
			System.out.println("Data is: "+prod.sb);
		}
	}
}
class Communciate
{
	public static void main(String[] args)
	{
		Producer p=new Producer();
		Consumer c=new Consumer(p);
		Thread t1=new Thread(p);
		Thread t2=new Thread(c);
```

```
            t2.start(); //Consumer thread will start first
            t1.start();
      }
}
```

Output:

```
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Appending
Data is: 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 :
```

Thread Priorities:

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as pre-emptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Program:

```
class TestMultiPriority extends Thread
{
      public void run()
      {
            Thread t=Thread.currentThread();
            System.out.println("running thread name is:"+t.getName());
            System.out.println("running thread priority is:"+t.getPriority());
      }
      public static void main(String args[])
      {
            TestMultiPriority m1=new TestMultiPriority();
            TestMultiPriority m2=new TestMultiPriority();
            Thread t1=new Thread(m1);
            Thread t2=new Thread(m2);
            t1.setPriority(Thread.MIN_PRIORITY);
            t2.setPriority(Thread.MAX_PRIORITY);
            t1.start();
            t2.start();
      }
}
```

Output:

```
running thread name is:Thread-3
running thread name is:Thread-2
running thread priority is:10
running thread priority is:1
```

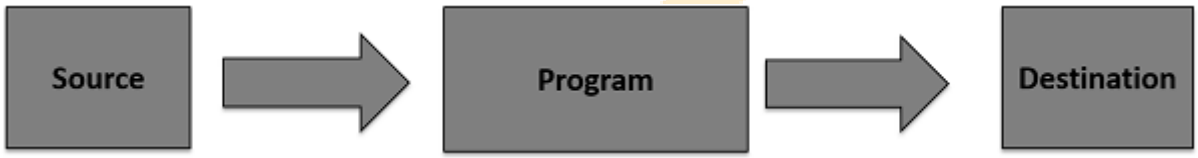
File operations in java:

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InputStream** − The InputStream is used to read data from a source.
- **OutputStream** − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams:

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;
public class CopyFile
{
   public static void main(String args[]) throws IOException
   {
      FileInputStream in = null;
      FileOutputStream out = null;
      try
      {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");
         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }
      finally {
         if (in != null)
             in.close();
         if (out != null)
            out.close();
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt.

Character Streams:

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit Unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having Unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile
{
   public static void main(String args[]) throws IOException
   {
      FileReader in = null;
      FileWriter out = null;
      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");
         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }
      finally
      {
         if (in != null)
            in.close();
         if (out != null)
            out.close();
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt

Standard Streams:

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three

standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

Example

```
import java.io.*;
public class ReadConsole
{
   public static void main(String args[]) throws IOException
   {
      InputStreamReader cin = null;
      try
      {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }
      finally {
         if (cin != null)
            cin.close();
      }
   }
}
```

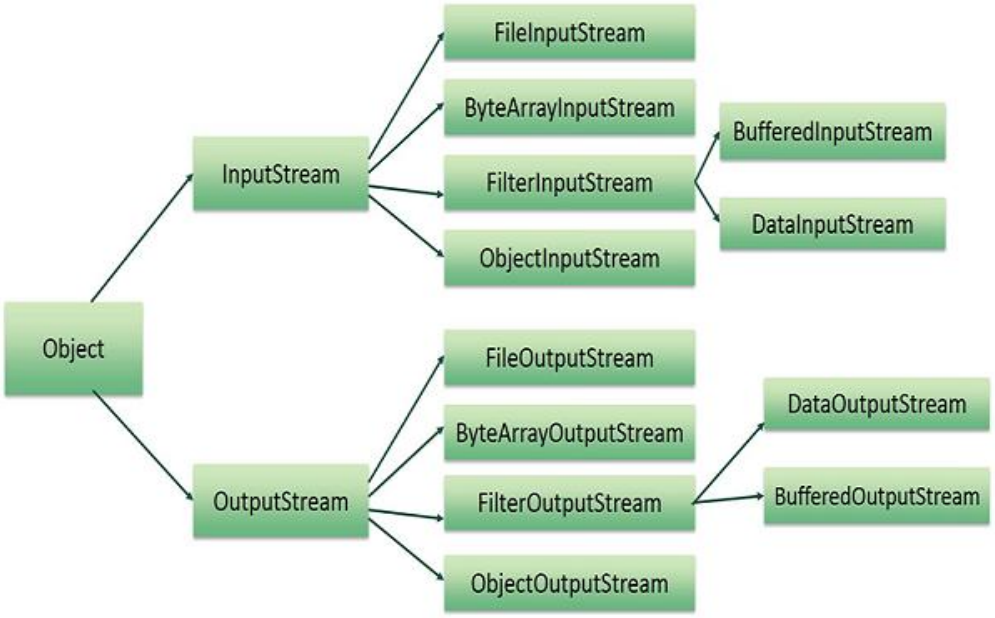
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
e
e
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is

	the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links −

- ByteArrayInputStream
- DataInputStream

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.