MODULE-5

Multi-Threaded Programming: The java thread model, Thread Life Cycle, The main() thread, creating a Thread, Creating Multiple Threads, Using isalive() and join(), Thread Priorities, Synchronization.
I/OFiles: Byte Oriented and Character oriented classes, Random Access Files. **Applets:** Introduction to Applets, Applet Life Cycle methods.

The Java Thread Model

A Thread is similar to a program that has single flow of control. It has a beginning, a body and an end and execute commands sequentially. All the main programs in our earlier examples can be called single threaded programs. Every program will have at least one thread as shown below.

```
class Chakry        ➔ Begin
{
.....
.....               ➔ Single Threaded body of
.....                 execution
....
..
}                   ➔ End
```

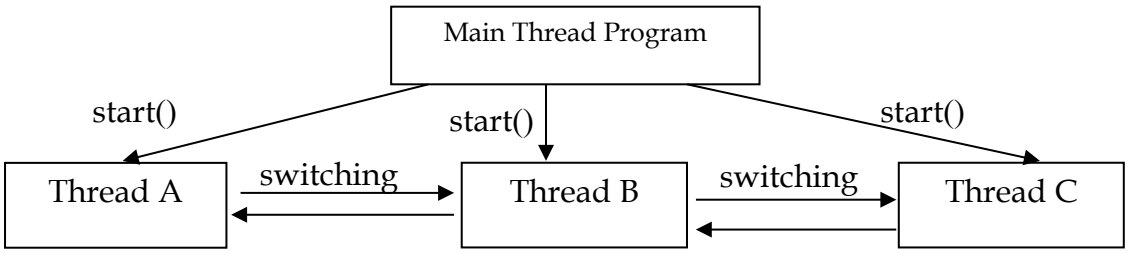
A program that contains multiple flows of control is known as multithreaded program. The following diagram shows a java program with four threads, one main thread and three others.

What is **Thread**?

- A thread is a single sequential flow of control within a program.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Multithreading in Java is a process of executing multiple threads simultaneously.
- Multithreading use a shared memory area, they don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- It differs from a "process" in that a process is a program executing in its own address space whereas a thread is a single stream of execution within a process.

Java Multithreading is mostly used in games, animation, etc.

Main Thread



The main thread is actually the main() method module, which is designed to create and start the other three threads namely A,B and C. Once initiated by the main() thread, the threads A,B and C run concurrently and share the resources jointly. Since threads in java are sub programs of a main application program and share the same memory space, they are known as light weight threads or light weight processes.

In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running. The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. All other threads continue to run. It is important to understand that Java's multithreading features work in both types of systems. In a single core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, **two or more threads do not actually run at the same time**, but idle CPU time is utilized. However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

Threads exist in several states. Here is a general description. A thread can be ***running***. It can be ***ready to run***as soon as it gets CPU time. A running thread can be ***suspended*****,** which temporarily halts its activity. A suspended thread can then be ***resumed***, allowing it to pick up where it left off. A thread can be

blockedwhen waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface: Java's multithreading system is built upon the Thread class, its methods, and another one is Runnable interface. Thread encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it. To create a new thread, your program will either extend Thread or implement the Runnable interface. The Thread class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

Method Meaning

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

Multithreaded Programming is a simple form of parallel processing in which several programs are run at the same time on a uniprocessor. Since there is only one processor, there can be no true simultaneous execution of different programs. Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time.

Multitasking: Multitasking in an operating system is allowing a user to perform more than one computer task (such as the operation of an application program) at a time. The operating system is able to keep track of where you are in these tasks and go from one to the other without losing information.

Multithreading: Multithreading is the ability of a program to manage its use by more than one thread at a time .Dispatchable atomic units of the program are executing simultaneously.

Multithreaded applications deliver their potent power by running many threads concurrently within a single program. From a logical point of view, multithreading means multiple lines of a single program can be executed at the same time, however, it is not the same as starting a program twice and saying that there are multiple lines of a program being executed at the same time. In this case, the operating system is treating the programs as two separate and distinct processes.

Uses of Threads:

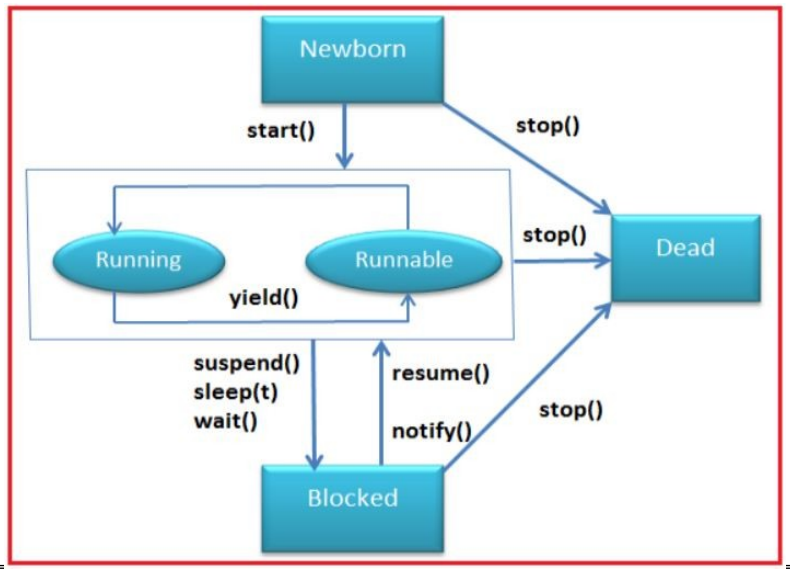
- Threads are used in designing server side programs to handle multiple clients at a time.
- Threads are used in games and animations.
- We can reduce the idle time of processor.
- Performance of processor is improved.
- Reduces interferences between execution and user interface.
-

LIFE CYCLE OF A THREAD

During the life time of a thread, there are many states it can enter. They include

1.New born state
2.Runnable state
3.Running state
4.Blocked state
5.Dead state

A thread is always in one of these 5 states. It can move from one state to another in many ways as shown below.



1. **New born state**: In this phase, the thread is created using class "Thread class". It remains in this state till the program starts the thread. It is also known as born thread.
2. **Runnable state**: In this state, the instance of the thread is invoked with a start() method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
3. **Running state**: When the thread starts executing, then the state is changed to "running" state. The scheduler selects one thread from the thread pool, and it starts executing in the application.
4. **Waiting(Blocked) state**: This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
5. **Dead state**: This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in "dead state".

The program uses the following methods **yield( ), sleep( )** and **stop( ).**

Program:

```
class A extends Thread
{
      public void run ( )
      {
           for(int  i=1; i<=20; i++)
           {
                if (i==3)
                yield ( );
                System.out.println("From thread A: i= "+i);
           }
       }
}
class B extends Thread
{
   public  void  run ( )
   {
      for (int j=1;j<=20;j++)
      {
          if (j==10)
          stop ( );
          System.out.println("From thread B: j="+j);
       }
   }
}
class C extends Thread
 {
   public void run ( )
   {
      for (int k=1;k<=20;k++)
      {
         try
         {
```

```
            sleep(1000);
          }
          catch (Exception e) { }
          System.out.println("From thread C: k="+k);
        }
    }
}
class ThreadMethods
 {
    public static void main(String args[ ])
    {
        A  p=new A( );
        B  q=new B( );
        C   r=new C( );
        p.start( );
        q.start( );
        r.start( );
     }
}
```

The Main() Thread

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

1. It is the thread from which other "child" threads will be CREATE.
2. Often, it must be the last thread to finish execution because it performs to close various actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. So, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown here:

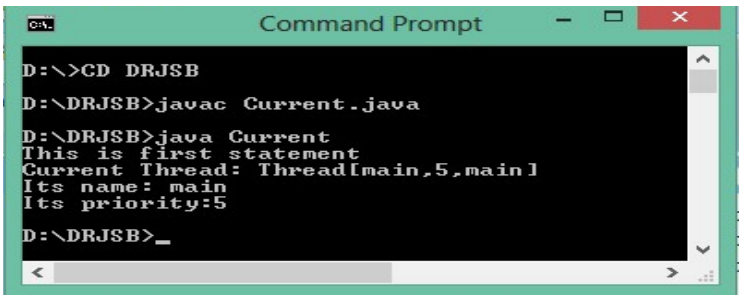
static Thread currentThread( )

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

Program : Write a program to know the currently running Thread

```
//Currently running thread
class Current
{
        public static void main(String args[])
        {
                        System.out.println ("This is first statement");
                        Thread t = Thread.currentThread ();
                        System.out.println ("Current Thread: " + t);
                        System.out.println ("Its name: " + t.getName ());
                        System.out.println ("Its priority:" + t.getPriority ());
        }
}
```

Output:



Creating Thread

A new thread can be created in two ways.

1. By creating a thread class
2. By converting a class to a thread

By creating a thread class

Define a class that extends thread class and override its run() method with the code required by the thread.

By converting a class to a thread

Define a class that implements **Runnable** interface. The **Runnable** interface has only one method, run( ), i.e., to be defined in the method with the code to be executed by the thread.

I. EXTENDING THE THREAD CLASS

We can make our class runnable as thread by extending the **class java.lang. Thread**. This allows us to access all the thread methods directly. It includes the following steps.

1. Declare the class as extending the **Thread** class.
2. Implement the **run()** method .
3. Create a thread object and call the **start()**.

Declaring the thread class:

The thread can be extended as follows

```
class MyThread  extends  Thread
 {
 …….……
       ……..
 }
```

Now we have a new type of thread **MyThread**.

Implementing the run ( ) method

The run ( ) method has been inherited by the class 'MyThread'. We have to override this method in order to implement the code to be executed by our thread. The basic implementation of run( ) is as follows

```
public void run()
{
   ……..
   ……..      // Thread code here
}
```

When we start the new thread, java calls the threads run( ) method.

Starting New Thread: To create and run an instance of our thread class we must write the following.

```
MyThread   aThread=new MyThread ( );   //Creation
           aThread.start( );           //invokes run( ) method
```

The thread is also started by using the following statement.

```
new MyThread ( ).start( );
```

STOPPING AND BLOCKING A THREAD

Stopping a Thread

Whenever we want to stop a thread from running further, we may do by calling its **stop( )** method.

```
aThread.stop( );
```

Here **aThread** is a thread object.

This statement causes the thread to move to the **Dead state**. A thread will also move to the dead state automatically when it reaches the end of its method. The **stop( )** method may be used when the thread is to be stopped before its completion.

Blocking a thread

A thread can also be temporarily suspended (or) blocked from entering into the runnable and running state by using the following thread methods.

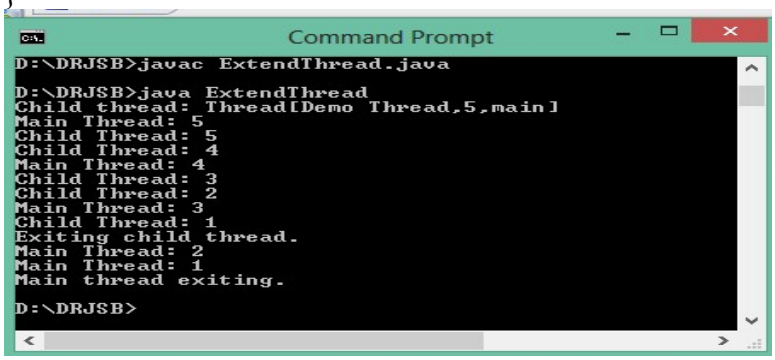
```
sleep ( )      // blocked for a specified time.
suspend ( )    // blocked until further orders.
wait ( )       // blocked until certain conditions occur.
```

These methods cause the thread to go into the blocked state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep ( ),** the **resume ( )** method is invoked in the case of **suspend ( )**, and the **notify ( )** method is called in the case of **wait( )** method.

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

```
// Create a second thread by extending Thread
class NewThread extends Thread {
        NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
```

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



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

public Thread(String *threadName*)

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

IMPLEMENTING THE RUNNABLE INTERFACE

The second method to create the threads is by using the **Runnable** interface. The **Runnable** interface declares the **run( )** method that is required for implementing threads in our programs. To do this, we must perform the following steps.

(1) Declare the class as implementing the **Runnable** interface
(2) Implementing the **run( )** method
(3) Create a thread by defining an object that is instantiated from the runnable class as the target of the thread.
(4) Calls the thread's **start( )** method to run the thread.

The following program illustrate the implementation of the above steps.

Program

```
class A implements Runnable
{
        public void run ( )
```

```
            {
                for (int i=1;i<=20;i++)
                System.out.println("Sri");
            }
    }
    class  RunnableEx
    {
        public static void main(String args[])
        {
            A  runnable=new  A( );
            Thread  t1=new Thread (runnable);
            t1.start ( );
        }
}
```

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;
        NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        }
// This is the entry point for thread.
public void run() {
        try {
        for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
        System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
        }
        }
class MultiThreadDemo {
public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
        // wait for other threads to end
        Thread.sleep(10000);
        } catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
        }
        }
```

output

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
```

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

As you see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main( )**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive( ) and join( )

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

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here: final void join( ) throws InterruptedException This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

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

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

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

output

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

As you can see, after the calls to **join( )** return, the threads have stopped executing.

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be to run. In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one.

For instance, when a lower-priority thread is running and a higher priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread. In theory, threads of equal priority should get equal access to the CPU. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a non preemptive operating system. In practice, even in non preemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O.

When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such

threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run. To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**. This is its general form:

final void setPriority(int *level*)

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

final int getPriority( )

The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

Example:

```
import java.lang.*;
  public class ThreadPriorityExample extends Thread   {

public void run()
{
System.out.println("Inside the run() method");
}

public static void main(String argvs[])
{
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);

System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
Thread.currentThread().setPriority(10);
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
}
}
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the

locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. You can synchronize your code in either of two ways. Both involve the use of the
synchronized keyword, and both are examined here.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method. To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**,

which pauses the current thread for one second. The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

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

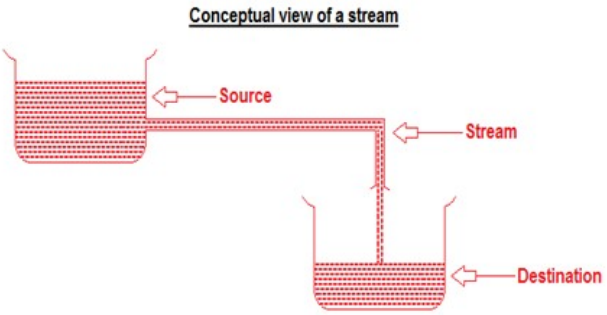
}
Here is the output produced by this program:
Hello[Synchronized[World]
]
]
As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used **sleep( )** to make the effects repeatable and obvious.To fix the preceding program, you must *serialize* access to **call( )**. That is, you must
restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

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

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

[Hello]
[Synchronized]
[World]

Streams: I/O Files



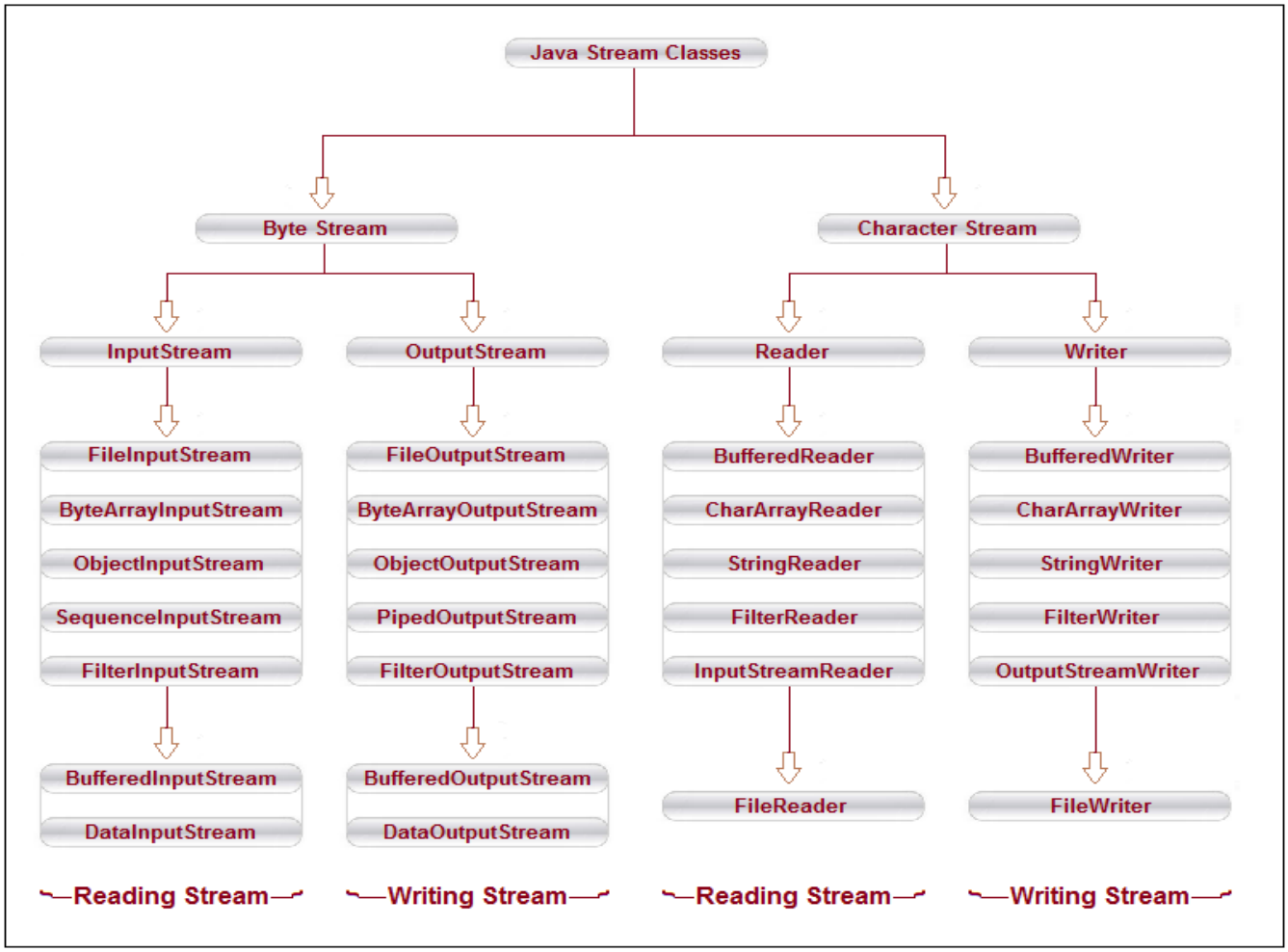
A stream carries data just as a water pipe carries water from one place to another. Streams can be categorized as input streams and output streams. Input streams are the streams which receive or read data while output streams are the streams which send or write data. All the streams are represented by classes in java.io (input – output) package.
Now, let us see how a stream works. We know an input stream reads data. So, to read data from keyboard, we can attach the keyboard to an input stream, so that the stream can read the data typed on the keyboard.

```
DataInputStream dis=new DataInputStream(System.in);
```

In the above statement, we are attaching the keyboard to DataInputStream object. The keyboard is represented by System.in. Now, DataInputStream object can read data coming from the keyboard. Here, System is a class and in is a field in System class. In fact, the System class has the following 3 fields:

- **System.in:** represents InputStream object. This object represents the standard input device, that is keyboard by default.
- **System.out:** represents PrintStream object. This object by default represents the standard output device, which is monitor.
- **System.err:** represents PrintStream object. This object by default represents the standard output device, which is monitor.

Types of java streams:



In java we are allowed to send data through streams only, either in the format of bytes or characters. Hence, based on the type of data passed, streams are divided into two types:

Byte streams and character streams:

Java defines two types of streams: byte and character. Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters.

Byte stream classes:

Byte streams represent data in the form of individual bytes. If a class name ends with the word 'stream', then it comes under byte streams. Byte streams are used to handle any characters (text), images, audio, and video files. For example, to store an image file (.jpg or .gif), we should go for a byte stream. Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers.

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read()** and **write(),** which respectively, read and write bytes of data. Both methods are declared as abstract inside **InputStream** and **OutputStream.** They are overridden by derived classes.

Character stream classes:

Character or text streams can always store and retrieve data in the form of characters (or text) only. It means text streams are more suitable for handling text files like the ones we create in notepad. They are not suitable to handle the images, audio, or video files. Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer**. Java has several concrete sub classes of each of these.

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

Types of streams:

Generally streams are divided into two types based on data flow direction.

InputStream: The stream that allows data to come into the java application from the persistent media is called input stream.

OutputStream: The stream that allows data to send out from the java application to be stored into the persistent media is called output stream. Basically input streams are used to read data from a persistent media and output streams are used to write or store data in persistent media from a java application.

FileInputStream and FileOutputStream: These classes are used to read and write data as bytes from file. Basically, these two streams are used as basic data source and destination for other streams.

DataInputStream and DataOutputStream: These classes are used to read and write data as primitive data. Basically, these two streams are used to add capability to another input stream and output stream inorder to read and write data as primitive types.

ObjectInputStream and ObjectOutputStream: These classes are used to read and write data as object. Basically, these two streams perform object serialization and deserialization.

BufferedInputStream and BufferedOutputStream: These classes are used to read and write data as buffers. Basically, these two streams are used to improve reading and writing performance of other streams.

SequenceInputStream: This class is used to read data from multiple Inputstreams sequentially.

PrintStream: This class is used to write data.

FileReader and FileWriter: These two classes are used to read and write data from a file as characters. It is recommended to use FileReader and FileWriter classes to read data from text file where as FileInputStream and FileOutputStream classes are only recommended to read and write image files.

BufferedReader and BufferedWriter: These two classes are used to improve reading and writing capability of other input and output stream.

InputStreamReader and OutputStreamWriter: An InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset.

An OutputStreamWriter is a bridge from character streams to byte streams. Characters written to it are encoded into bytes using a specified charset.

Binary input/output:

FileInputStream and FileOutputStream: These classes are used to read and write data as bytes from file. Basically, these two streams are used as basic data source and destination for other streams.

FileOutputStream: This class is used to write data to a file.

Constructors:

1. FileOutputStream(String name) throws FileNotFoundException
2. FileOutputStream(File f) throws FileNotFoundException
3. FileOutputStream(String name, boolean append) throws FileNotFoundException
4. FileOutputStream(File f, boolean append) throws FileNotFoundException

Creating a file using FileOutputStream:

FileOutputStream class belongs to byte stream and stores data in the form of individual bytes. It can be used to create text files. The following steps are to be followed to create a text file that stores some characters.

1. First os all, we should read data from the keyboard. For this purpose, we should attach keyboard to some input stream class. The code for using **DataInputStream** class for reading from the keyboard is as:

   DataInputStream dis=new DataInputStream(System.in);

   Here, **System.in** represents the keyboard which is linked with **DataInputStream** class object **dis**.
2. Now, attach a file where the data is to be stored to some output stream. Here, we take the help of **FileOutputStream** which can send data to the file. Attaching the file **myfile.txt** to **FileOutputStream** can be done as:

   FileOutputStream fout=new FileOutputStream("myfile.txt");
3. The next step to read data from **DataInputStream** and write it into **FileOutputStream** and write it into **FileOutputStream.** It means read data from dis object and write it into fout object, as shown here:

   char ch=(char)dis.read();
   fout.write(ch);

   Finally, any file should be closed after performing input or output operations on it, else the data of the file may be corrupted. Closing the file is done by closing the associated streams. For example, **fout.close();** will close the **FileOutputStream**.

E.g.:

```
//creating a text file using FileOutputStream
import java.io.*;
class FileCreate
{
public static void main(String args[])throws IOException
{
//attach keyboard to DataInputStream
```

```
DataInputStream dis=new DataInputStream(System.in);

//attach myfile to FileOutputStream
FileOutputStream fout=new FileOutputStream("myfile1.txt",true);

System.out.println("Enter text (@ at the end): ");
char ch;

//read characters from dis into ch. Then write them into fout.
//repeat this as long as the read character is not @
while((ch=(char)dis.read())!='@')
fout.write(ch);

//close the file
fout.close();
}
}
```

O/P: F:\JAVA>javac filecre.java
F:\JAVA>java FileCreate
Enter text (@ at the end):
this is vijay
@
F:\JAVA>type myfile1.txt
this is vijay
this is vijay

Improving efficiency using BufferedOutputStream;

If buffered classes are used, they provide a buffer (temporary block of memory), which is first filled with characters and then all the characters from the buffer can be at once written into the file. Buffered classes should be used always in connection to other stream classes. For example, BufferedOutputStream can be used along with FileOutputStream to write data into a file.

Thus, by using buffered classes, the speed of writing is improved. In the same way, we can use buffered classes for improving the speed of reading operation also.

We can attach FileOutputStream to BufferedOutputStream as:

```
BufferedOutputStream bout=new BufferedOutputStream(fout,1024);
```

Here, the buffer size is declared as 1024 bytes. If the buffer size is not specified, then a default buffer size of 512 bytes is used.

E.g.:

```
//creating a text file using BufferedOutputStream
import java.io.*;
class FileCreate
{
public static void main(String args[])throws IOException
{
//attach keyboard to DataInputStream
DataInputStream dis=new DataInputStream(System.in);

//attach myfile to FileOutputStream in append mode
FileOutputStream fout=new FileOutputStream("myfile.txt", true);

//attach FileOutputStream to BufferedOutputStream
BufferedOutputStream bout=new BufferedOutputStream(fout,1024);

System.out.println("Enter text (@ at the end): ");
char ch;

//read characters from dis into ch. Then write them into bout.
//repeat this as long as the read character is not @
while((ch=(char)dis.read())!='@')
bout.write(ch);

//close the file
```

```
bout.close();
}
}
```

O/P: F:\JAVA>javac filecre1.java

F:\JAVA>java FileCreate
Enter text (@ at the end):
this is II mca
this is necn
@

F:\JAVA>type myfile.txt
this is my file line four
this is my file line five
this is my file line six
this is my file line seven
this is my last line
this is Mukesh
this is mca
this is II mca
this is necn

Reading data from a file using FileInputStream:

FileInputStream: It must be used to read data from a file.
Constructors:

1. FileInputStream(String name) throws FileNotFoundException
2. FileInputStream(File f) throws FileNotFoundException
3. FileInputStream(FileDescriptor fd) throws FileNotFoundException

FileInputStream is useful to read data from a file in the form of sequence of bytes. It is possible to read data from a text file using **FileInputStream**.

- First, we should attach the file to a FileInputStream as shown below:

FileInputStream fin=new FileInputStream("myfile.txt");

This will enable us to read data from the file. Then, to read data from the file, we should read data from the FileInputStream as:

ch=fin.read();

- Then, we should attach the monitor to output stream, example PrintStream, so that the output stream will send data to the monitor. For displaying the data, we can use System.out which is nothing but PrintStream object.

System.out.print(ch);

E.g.:

```
//reading text file using FileInputStream
import java.io.*;
class FileRead
{
public static void main(String args[])throws IOException
{
//attach the file to FileInputStream
FileInputStream fin=new FileInputStream("myfile.txt");

System.out.println("File contents: ");

/*read characters from FileInputStream and write them
to moniter. Repeat this till the end of file*/
int ch;
while((ch=fin.read())!= -1)
System.out.print((char)ch);
//close the file
fin.close();
}
}
```

O/P: F:\JAVA>javac fileread.java

F:\JAVA>java FileRead
File contents:
this is my file line four
this is my file line five
this is my file line six
this is my file line seven
this is my last line
this is Mukesh
this is mca
this is II mca
this is necn

The above program works with myfile.txt only. To make this program work with any file, we should accept the file name from the keyboard. For this purpose, create BufferedReader object as:

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

Here, the keyboard (System.in) is attached to InputStreamReader which is attached to BufferedReader. So if we read data from the BufferedReader, then that is actually read from the keyboard. Using readLine() method of the BufferedReader class we can read the file name from the keyboard as:

String fname=br.readLine();

The file name should be attached to FileInputStream for reading data as:

FileInputStream fin=new FileInputStream(fname);

E.g.:

```
//reading data from any text file
import java.io.*;
class FileRead
{
public static void main(String args[])throws IOException
{
//to accept file name from keyboard
BufferedReader br=new BufferedReader
                    (new InputStreamReader(System.in));
System.out.print("Enter file name: ");
String fname=br.readLine();

//attach the file to FileInputStream
FileInputStream fin=null; //assign nothing to fin

//check if file exists or not
try
{
fin=new FileInputStream(fname);
}
catch(FileNotFoundException fe)
{
System.out.println("File not found");
return;
}
//attach FileInputStream to BufferedInputStream
BufferedInputStream bin=new BufferedInputStream(fin);
System.out.println("File contents: ");
/*read characters from BufferedInputStream and write them
to moniter. Repeat this till the end of file*/
int ch;
while((ch=bin.read())!= -1)
System.out.print((char)ch);
//close the file
bin.close();
}
}
```

O/P: F:\JAVA>javac fileread1.java

F:\JAVA>java FileRead
Enter file name: myfile315.txt
File contents:
this is necn

DataInputStream and DataOutputStream:

These classes are used to read and write the data as primitive type from the underlying input stream and output stream. These classes have special methods to perform reading and writing operation as primitive data type.

DataInputStream: It is a sub class of FilterInputStream and DataInput interface. It implements following methods from the DataInput interface for reading bytes from a BinaryInputStream and convert them into corresponding java primitive data types.

DataOutputStream: It is a sub class of FilterOutputStream and DataOutput interface. It implements following methods from the DataOutput interface for converting data from any of the java primitive data types to a stream of bytes and writing these bytes to a BinaryOutputStream.

To read data as primitive data types using readxxx() method, the data must be written to the file as primitive types using writexxx() method.

E.g.:

```
//reading and writing primitive data
import java.io.*;
class ReadWrite
{
public static void main(String args[])throws IOException
{
//File primitive=new File(prim.dat);
FileOutputStream fos=new FileOutputStream("prim.txt");
DataOutputStream dos=new DataOutputStream(fos);
//write primitive data to the "prim.txt" file
dos.writeInt(1999);
dos.writeDouble(375.12);
dos.writeBoolean(false);
dos.writeChar('V');
dos.close();
fos.close();
//read data from the "prim.txt" file
FileInputStream fis=new FileInputStream("prim.txt");
DataInputStream dis=new DataInputStream(fis);
System.out.println(dis.readInt());
System.out.println(dis.readDouble());
System.out.println(dis.readBoolean());
System.out.println(dis.readChar());
dis.close();
fis.close();
}
}
```

O/P: F:\JAVA>javac prim.java
F:\JAVA>java ReadWrite
1999
375.12
false
V

Text input/output

FileReader and FileWriter:

These two classes are used to read and write data from a file as characters. It is recommended to use FileReader and FileWriter classes to read data from text file where as FileInputStream and FileOutputStream classes are only recommended to read and write image files.

FileReader: FileReader is useful to read data in the form of characters from a text file.

Constructors:

1. FileReader(String name)
2. FileReader(File f)

Methods of FileReader;

1. int read()

It attempts to read next character from the file and returns its Unicode value. If the next character is not available, then we will get -1 i.e. it returns -1.

2. int read(char[] ch)

It attempts to read enough characters from the file into the char[] array and returns the number of characters copied.

3. void close()

E.g.:

```
//Reading data from a file using  FileReader
import java.io.*;
class ReadFile
{
public static void main(String args[])throws IOException
{
int ch;
//check if file exists or not
FileReader fr=null;
//check if file exists or not
try
{
fr=new FileReader("myfile.txt");
}
catch(FileNotFoundException fe)
{
System.out.println("File not found");
return;
}
//read from FileReader till the end of file
while((ch=fr.read())!=-1)
System.out.print((char)ch);
//close the file
fr.close();
}
}
```

O/P: F:\JAVA>javac filereader.java
F:\JAVA>java ReadFile
this is Mukesh

FileWriter: FileWriter is useful to create a file by writing characters into it.

Constructors:

1. FileWriter(String name)
2. FileWriter(File f)
3. FileWriter(String name, boolean append)
4. FileWriter(File f, boolean append)

Methods:

1. write(int i): To write a single character to the file.
2. write(char[] ch): To write an array of characters to the file.
3. write(String s): To write a string to the file.
4. flush(): To give guarantee that the last character of the data is also added to the file.
5. close(): Close the file.

E.g.:

```
//creating a file using  FileWriter
import java.io.*;
class CreateFile
{
public static void main(String args[])throws IOException
{
String str="This is a book on java."+"\n I am learner of java.";
//attach file to FileWriter
FileWriter fw=new FileWriter("new.txt");
//read characterwise from string and write it into FileWriter
for(int i=0;i<str.length();i++)
```

```
fw.write(str.charAt(i));
//close the file
fw.close();
}
}
```

O/P: F:\JAVA>javac filewriter.java
F:\JAVA>java CreateFile
F:\JAVA>type new.txt
This is a book on java.
I am learner of java.

BufferedReader and BufferedWriter:

These two classes are used to improve reading and writing capability of other input and output stream.

BufferedWriter: We can use the BufferedWriter to write character data to the file.

Constructors:

1. BufferedWriter bw=new BufferedWriter(writer w);
2. BufferedWriter bw=new BufferedWriter(writer w, int buffersize);

BufferedWriter never communicates directly with the file.

BufferedWriter bw=new BufferedWriter(new FileWriter("newfile1.txt"));

Methods:

1. write(int i)
2. write(char[] ch)
3. write(String s)
4. flush()
5. close()
6. newLine(): To insert line separator.

E.g.:

```
import java.io.*;
class Buffer
{
public static void main(String args[])throws IOException
{
BufferedWriter bw=new BufferedWriter(new FileWriter("new1.txt"));
bw.write(68);
bw.newLine();
char[] ch={'a','b','c'};
bw.write(ch);
bw.newLine();
bw.write("vbr");
bw.newLine();
bw.flush();
bw.close();
}
}
```

O/P: F:\JAVA>javac buff.java
F:\JAVA>java Buffer
F:\JAVA>type new1.txt
D
abc
Mukesh

BufferedReader: We can use the BufferedReader to read data from the file. The main advantage of BufferedReader over the FileReader is we can read the data line by line instead of character by character.

Constructors:

1. BufferedReader br=new BufferedReader(Reader r);
2. BufferedReader br=new BufferedReader(Reader r, int buffersize);;

Methods:

1. int read()
2. int read(char[] c)
3. void close();
4. String readLine()

E.g.:

```
import java.io.*;
class Buffer
{
public static void main(String args[])throws IOException
{
BufferedReader br=new BufferedReader(new FileReader("new1.txt"));
String line=br.readLine();
while(line!=null)
{
System.out.println(line);
line=br.readLine();
}
br.close();
}
}
```

O/P: F:\JAVA>javac buff.java
F:\JAVA>java Buffer
D
abc
Mukesh

File management using File class:

File class of java.io package provides some methods to know the properties of a file or directory. First of all, we should create the File class object by passing the file name or directory name to it.

File obj=new File(file name);
File obj=new File(directory name);
File obj=new File("path",file name);
File obj=new File("path",directory name);

File class methods:

File class includes the following methods:

- **boolean isFile():** This method returns true if the File object contains a file name, otherwise false.
- **boolean isDirectory:** This method returns true if the File object contains a directory name.
- **boolean canRead():**This method returns true if the File object contains a file which is readable.
- **boolean canWrite():**This method returns true if the file is writable.
- **boolean canExecute():**This method returns true if the file is executable.
- **boolean exists():**This method returns true when the File object contains a file or directory which physically exists in the computer.
- **String getParent():** This method returns the name of the parent directory of a file or directory.
- **String getPath():** This method returns the name of the directory path of a file or directory.
- **String getAbsolutePath():** This method returns the absolute directory path of a file or directory location.
- **long length():** This method returns a number that represents the size of the file in bytes.
- **boolean delete():** This method deletes the file or directory whose name is in File object.
- **boolean createNewFile():** This method automatically creates a new, empty file indicated by a File object, if and only if a file with this name does not yet exist.
- **boolean mkdir():** This method creates a directory whose name is given in File object.
- **boolean renameTo(File newname):** This method changes the name of the file as new name.
- **String []list():** This method returns an array of strings naming the files and directories in the directory.

E.g.:

```
import java.io.*;
class FileProp
{
    public static void main(String a[])
    {
        String fname=a[0];
        File f=new File(fname);
        System.out.println("File name:"+f.getName());
        System.out.println("path:"+f.getPath());
        System.out.println("Absolute path:"+f.getAbsolutePath());
```

```
        System.out.println("parent:"+f.getParent());
        System.out.println("Exists:"+f.exists());
        if(f.exists())
        {
            System.out.println("is writeable:"+f.canWrite());
            System.out.println("is readable:"+f.canRead());
             System.out.println("is a directory:"+f.isDirectory());
            System.out.println("FIle size in bytes:"+f.length());

    }
}
}
```

File copy: sometimes we need to copy the entire data of a text file into another text file. Streams are useful in this case. How to use streams for copying file content to another file, we can use the following logic:

1. For reading data from the input file, attach it to FileInputStream.
2. For writing data into the output file, which is to be created, attach it to FileOutputStream.
3. Now, read data from FileInputStream and write into FileOutputStream. This means, the data is read from the input file and send to output file.

E.g.:

```
//copying a file contents as another file
import java.io.*;
class CopyFile
{
public static void main(String args[])throws IOException
{
int ch;

//for reading data from args[0]
FileInputStream fin=new FileInputStream(args[0]);

//for writing data into args[1]
FileOutputStream fout=new FileOutputStream(args[1]);

//read from FileInputStream and write into FileOutputStream
while((ch=fin.read())!=-1)
fout.write(ch);

//close the files
fin.close();
fout.close();
System.out.println("1 file copied");
}
}
```

O/P: F:\JAVA>javac filecpy.java

F:\JAVA>java CopyFile filecpy.java filecpy1.java
1 file copied

F:\JAVA>type filecpy1.java

```
//copying a file contents as another file
import java.io.*;
class CopyFile
{
public static void main(String args[])throws IOException
{
int ch;

//for reading data from args[0]
FileInputStream fin=new FileInputStream(args[0]);
```

```
//for writing data into args[1]
FileOutputStream fout=new FileOutputStream(args[1]);

//read from FileInputStream and write into FileOutputStream
while((ch=fin.read())!=-1)
fout.write(ch);

//close the files
fin.close();
fout.close();
System.out.println("1 file copied");
}
}
```

Random access file operations

Sequential files can read / write only at the beginning / ending of the file. On the other hand, random access files allow us to read from or write to any location in the file. The class RandomAccessFile offers methods that allow specified mode accesses such as 'read only' and 'read - write' to files. Since RandomAccessFile is not inherited from InputStream or OutputStream. It implements both DataInput and DataOutput which deal with java's primitive data types. The RandomAccessFile class is a very useful class for file handling. The constructor of these classes is the following:

RandomAccessFile(String filename, String mode);
RandomAccessFile(File f, String mode);

Thus, RandomAccessFile objects can be created either from a string containing the file name or from a File object. The mode represents the type of access to the file, for example, 'r' for read and 'rw' for read and write. There are two other modes, namely, 'rws' and 'rwd', 'rws' opens the file as read and write and stores the updations made on the file data or metadata in the synchronized way. Whereas, 'rwd' deals with the file data but not the meta data. Here meta data refers to the data about the files. The method seek() points the file pointer to the location specified by number. RandomAccessFile throws an IOException if an I/O error has occurred.

E.g.:

```
//writing and reading with random access
import java.io.*;
class Random
{
public static void main(String args[])throws IOException
{
RandomAccessFile file=null;
file=new RandomAccessFile("random.txt","rw");
file.writeChar('V');
file.writeInt(2012);
file.writeDouble(3.14);

file.seek(0); //go to the begining

//reading from the file
System.out.println(file.readChar());
System.out.println(file.readInt());
System.out.println(file.readDouble());

file.seek(2); //go to the second item
System.out.println(file.readInt());

//go to the end and append true to the file
file.seek(file.length());
file.writeBoolean(true);
file.close();
}
}
```

O/P: E:\Mukesh\JAVA>javac random.java

E:\Mukesh\JAVA>java Random
V
2012
3.14
2012

Introduction to Applet

Applets: When an HTML page wants to communicate with the user on internet, it can use a special java program, called 'applet' to communicate and respond to the user. The user can interact by typing some details or by clicking the components available in the applet program. The program then processes and displays the results. An applet as a java byte code embedded in a HTML page.

One of the main features of java is the applet. Applets are dynamic and interactive programs.

Applets are small java programs that are basically used in internet programming. They can be transported over the internet from one computer to another and run using appletviewer" (or)any web browsers that supports java .An applet can perform the following tasks.

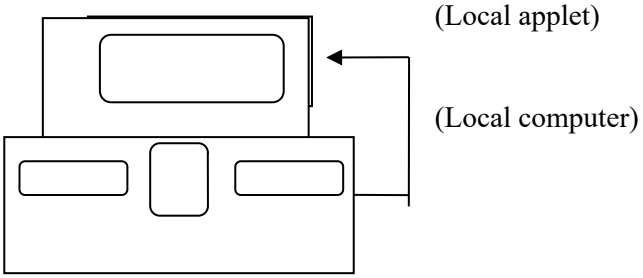
Arithmetic operations, play sounds, display graphics, create animation, accept user input and interactive games.

We can embed applets into web pages in two ways. They are

1. Local applets
2. Remote applets

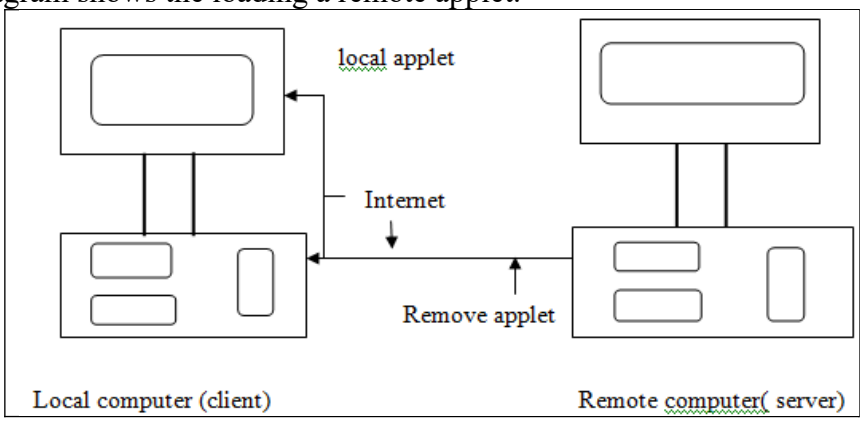
Local applets: An applet developed locally and stored in a local system is known as a "local applet". We can write our own applets and embedded them into the web pages, local applet does not need to use the internet. It simply searches the directories in the local system and locates and loads the specified applet.

The following shows the loading of local applet.



Remote applet: A remote applet is that which is developed by some one else and stored on a remote computer connected to the internet. If our system is connected to the internet we can download the Remote applet on to our system via at the internet and run it.

The following diagram shows the loading a remote applet.



In order to locate and load a remove applet ,we must know the applet address on the web. This address is known as "Uniform Resource Locater"(URL) and must be specified in the applets HTML documents as the value of the CODE BASE ATTRIBUTE

CODE BASE = http://www.netserve.com/applets

How Applets Differ From Applications:

Both the applets and standard alone applications are java programs there are significant differences between them they are:-

1. Applet do not use the main() method for start the execution of the code. Applets when loaded automatically call certain methods of "Applet" Class to start and execute the applet code.
2. Applets are executed from inside a webpage using a special feature known as HTML Tag.

3. Applet cannot read from or write to the files in the local computer.
4. Applets cannot run any program from the local computer.
5. Applets are restricted from using libraries from other languages such as C or C++.
6. Applets cant communicate with other servers on the network.

Java applications	Java applets
1. These run on standalone systems. 2. These run from the command line. 3. Parameters to the application are given at the command prompt for example, args[0], args[1], and so on. 4. In an application, the program starts at the main() method. The main() method runs throughout the application. 5. These are run by specifying at the command prompt as follows: java classfilename	1. These run in web pages. 2. These are executed using a web browser. 3. Parameters to the applet are given in the HTML file. 4. In an applet, there is no main() method. 5. These are run by specifying the URL in a web browser, which opens the HTML file. Or run using the appletviewer by specifying at the command prompt: **appletviewer htmlfilename**

Preparing To Write Applets: The following are the situations when we need to use applets.
1. When we need something dynamic to be included in the display of a webpage.
2. When we requires some flash outputs.
3. When we want to create a program and make it available on the internet for us by others on their computers.

The steps involved in developing and in testing are:-

- Building an applet code(. java file)
- Creating an executable applet (. class file)
- Designing a web page using HTML Tags
- Preparing < applet > Tag
- Incorporating < applet > tag into the webpage.
- Creating HTML file(.html file)
- testing the applet code

Building an applet code: An applet code uses the services of two classes :

1. Applets (java.applet package)
2. Graphics (java.awt package)

The **Applet** class which is contained in the "java.applet" package provides life and behaviour to the applet through its methods such as **init(),start() and paint().**When an applet is loaded java automatically calls a series of Applet class methods for starting, Running and stopping the applet code. The applet class maintains the life cycle of a applet.

The paint() method of the applet class, is used to displays the result of the applet code on the screen. The output may be text, graphics (or) sound. The paint() method which requires a **"Graphics"** object as an argument is defined as follows.

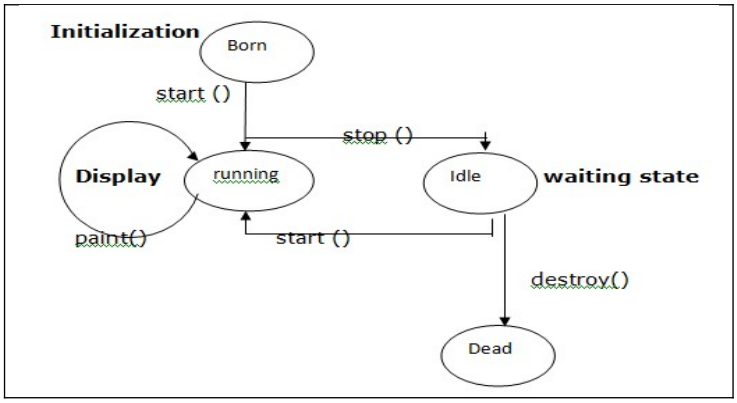
```
public void paint(Graphics  g)
{--------}
```

Ex: Program:-

```
import java.awt.*;
import java.applet.*;
public void HelloJava extends Applet
{
        public void paint (Graphics g)
                {
                        g.drawString ("Hello Java",10,100);
                }
}
```

Applet Life Cycle /Methods

Every java applet inherits a set of default behaviours(methods) from the Applet class. When an applet is loaded, enters into different states. The following diagrarm shows the applet states.



The applet states includes:

- Born on initialization state
- Running
- Idle state
- dead or destroyed state

Initialization: -

Applet enters the initialization state when it is first loaded. This is achieved by calling the init( ) method of Applet class. The applet is born at this stage we may perform the following things , if required

- Create objects needed by the applet
- Set up initial values
- Load images or fonts
- Set up colors

The initialization occurs only once in the applet Life cycle. To provide any of the behaviours mentioned above, we must Override the init() method.

```
public void init()
{
------
------    (Action)
}
```

Running state:-Applet enters the "Running state" when the system calls the start() method of Applet class. This occurs automatically after the applet is initialized. starting can also occur if the already in "stopped" (idle) state.

```
public void start()
{
---------
---------   (action)
}
```

Idle (or) Stopped state:- An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page currently running applet. We can also calling enter in to the idle state by calling stop() method explicitly. If we use a thread to run the applet, then we must use stop() method to terminate the thread. We can achieve this by overriding the stop() method.

```
public void stop()
{
-------
-------      (action)
}
```

Dead state:-

An applet is said to be "dead" state , when it is removed from memory. This occurs automatically by invoking the destroy( ) method .This occurs automatically when we quit the browser. Dead state also occurs only once in the applet Life cycle. If the applet has created any resources like threads, we may overrides the destroy( ) method to clean up these resources.

```
public void destroy()
{
-------
-------      (action)
}
```

Display state:-

Applet moves to the display state whenever it has to perform some output operations on the screen. This happens immediately after the applet enters into the running state. The paint() method is called to

perform these task. Almost every applet will have a paint() method. We must override this method if we went anything to be displayed on the screen.

```
public  void paint (Graphics g)
{
- - - - - - - - -
- - - - - - - - -     (display statements)
}
```

2.Creating a executable Applet: -

Executable appplet is the ".class" file of the applet which is obtained by compiling the source code of the applet. We can use the java compiler to compile the applet.

The following are the steps required for compiling "HelloJava"Applet.

a) Move to the directory containing the source code and type the following command.

javac HelloJava.java

b) The compiler output file called "HelloJava.class" is placed in the same directory as the source.

c) If any error message is received, then we must check for errors, correct them, and compile the applet again.

3.Designing a webpage:-

Java applets are programs that are stored as an web pages. In order to run a java applet, it is first necessary to have a webpage that references the applet.

A webpage is made up of TEXT and HTML tags that can be interpreted by a Browser (or) an "appletviewer". A web page is designed by using any text editor. A webpage is also known as HTML page or HTML document. Web pages are stored using a file extension". html" such as "hellojava.html" .HTML file should be stored in the same directory as the compiled code of the applets.

Web pages includes text that we want to display is HTML Tags (command) to web browsers. A web pages is marked by an opening html tag

< HTML > and a closing html tag </ HTML > and is divided into the following three major sections.

1. Comment section (optional)
2. Head section (optional)
3. Body section

1. **Comment section:-**This section contains comments about the webpage. A comment line begin with a

<! " Ends with a " >

web browsers will ignore the text enclosed between them.

2. **Head section:-**The head section is defined with a starting < Head > tag and a closing < /HEAD > tag. This section usually contains a title for the webpage as shown below.

```
< HEAD >
    < TITLE > welcome to java applets < / TITLE >
< / HEAD >
```

Text will appear in the title of the web browsers when it displays the page. The head section is also optional.

3. **Body section**:-It is the third section of an webpage which contains the entire information about the webpage and its behaviour. We can set up many potions to indicate how our page must appear on the screen (like colour, location, sound etc..) shown below is a simple body section.

```
< Body >
    < CENTER >
        < HI > welcome to the world of applets < H >
    < / CENTER >
    < APPLET… >
    < / APPLET
< / Body >
```

4.Applet Tag:-An applet tag is used to embedded the executable applet into the webpage. It used the following system.

```
< APPLET
    [ CODE BASE = code base – URL]
  CODE = Applet Filename.class
    [ ALT =  alternate – text ]
    [ NAME = applet – instance - name ]
  WIDTH = pixels
  HEIGHT = pixels
    [ ALIGN = alignment ]
    [VSPACE = pixels ]
    [HSPACE = pixels ]   >
    [  < PARAM   NAME  = name 1 VALUE = value 1 > ]
```

[< PARAM NAME = name 2 VALUE = value 2 >]
- - - - - - - - -
- - - - - - - - -
[Text to be displayed in the absence of java]
< / APPLET >

The following table shows the list attributes and their meaning.

ATTRIBUTE	MEANING
1. Code = Applet Filename. Class	Specifies the name of the applet class to be loaded. That is, the name of the Applet already compiled.
2. CODEBASE = code base_ URL (optional)	Specifies the URL of the directory in which the applet is stored.
WIDTH = pixels HIGHT = pixels	These attribute specify the width and height of the space on the HTML Page that will be reserved for the applet
3. Name = applet instance-name(optional)	The name of the applet maybe specified so that other applets on the page may refer to this applet.
4. ALIGN = alignment (optional)	This specifies where the applet is placed on web page.
5. HSPACE = pixels (optional)	This attribute specifies the amount of horizontal blank space the browser should leave surrounding the applet.
6. VSSPACE = pixels (optional)	This specifies the amount of vertical blank space the browser should leave surrounding the applet.
7. ALT = alternate - text (optional)	Non-java browsers will display this text when the applets are handled.

5.Adding applet to html file:-

We can insert the < Applet > tag in the page at the place where the output of the applet must appear. Following is the content of the HTML file that is embedded with the < APPLET > tag of our "HelloJava" applet.

```
< HTML >
        <  BODY  >
        <  APPLET
                CODE = HelloJava.class
                 WIDTH = 400
                 HEIGHT = 200>
        < / APPLET >
        < / BODY >
< / HTML >
```

6.Running the Applet:-

We must check the following files in our current directory:

HelloJava.java
HelloJava.class
HelloJava.html

To run an applet, we require one of the following tools:

1. Java-enabled Web browser (such as HotJava or Netscape)
2. Java appletviewer

If we use a Java-enabled Web browser we will be able to see the entire Web page containing the applet.
The appletviewer is available as a part of the java development kit(jdk) .if we use "applet viewer "tool we will see only the Applet.

Syn:- appletviewer HelloJava.html

Passing Parameters to Applet

We can supply user-defined parameters to an applet using <PARAM…> tags. Each <PARAM…> tag specifies two attributes such as "name", and "value" attribute. "name" specifies name of the parameter and "value" specifies value of the variable. <PARAM…> tag as follows:

```
<APPLET . . . .>
<PARAM NAME = text VALUE = " I love Java " >
< / APPLET>
```

Passing parameters to an applet code using <PARAM> tag is something similar to passing parameters to the main () method using command line arguments. To set up and handle parameters, we need to do two things:

1. Include appropriate <PARAM . . .> tags in the HTML document.

2. Provide code in the applet to parse these parameters.

In java code ,"getParameters()" method, is used to access the parameter values.

Program:

```
import java.awt,*;
import java.applet.*;
public class HelloJava extends Applet
{
 String str;
 public void init ()
 {
str = getParameter("string");  //Receiving parameter value
if  (str == null )
str  =  " Java ";
str  =  " Hello " +str;                  // using the value
    }
 public void paint (Graphics g)
 {
g.drawString(str, 10, 100);
 }
  }
<HTML>
 < !  Parameterized  HTML  file  >
 <HEAD>
 <TITLE> Welcome to Java Applets  < / TITLE>
 </HEAD>
 <BODY>
  <APPLET  CODE  =  HelloJava.class
   WIDTH = 400
   HEIGHT = 200>
  <PARAM  NAME = " string"
   VALUE  = " Applet ! ">
  < / APPLET>
 < / BODY>
< / HTML>
```

Aligning the Display:-

We can align the output of the applet using the "ALIGN" attribute. This attribute can have one of the nine values:

LEFT, RIGHT, TOP, TEXT TOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM.

For example, A LIGN = LEFT

will display the output at the left margin of the page. All text thatFollows the ALIGN in the Web page will be placed to the right of the display.

The following program shows a HTML file for our HelloJava applet with ALIGN attribute.

```
<HTML>
 <HEAD>
 <TITLE> Here is an applet < / TITLE>
 < / HEAD>
 <BODY>
  <APPLET CODE  = HelloJava.class
   WIDTH  = 400  HEIGHT = 200
   ALIGN   = RIGHT >
  < / APPLET>
 < / BODY>
< / HTML>
```

Displaying Numerical Values:-

In applets, we can display numerical values by first converting them into strings and then using the drawstring() method of "Graphics" class. We can do this easily by calling the valueOf() method of "String" class.

Displaying numerical values:-

```
import  java.awt.*;
```

```
import java.applet.*;
public class NumValues extends Applet
{
public void paint (Graphics g)
        {
                int value1 = 10;
                int value2 = 20;
                int sum = value1 + value2;
                String  s = " sum: "+ String.valueOf(sum);
                g.drawString(s,100,100);
        }
 }
<html>
<applet code = NumValues.class
        Width = 300
        Height = 300 >
< / applet>
< / html>
```

E.g.2 :

```
import java.applet.*;
import java.awt.*;
/*<applet code="sampleapplet.class" width=300 height=300>
</applet>*/
public class sampleapplet extends Applet{
String msg=" ";
public void init()
{
msg=msg+"init();";
}
public void start()
{
msg=msg+"start();";
}
public void destroy()
{
msg=msg+"destroy();";
}
public void stop()
{
msg=msg+"stop();";
}
public void paint(Graphics g)
{
msg=msg+"paint();";
g.drawString(msg,10,20);
}
}
```

Example 3:

```
import java.awt.*;
import java.applet.*;
public class HelloJava extends Applet{
public void init()
{
System.out.println("this is init() method");
}
public void start()
{
System.out.println("this is start() method");
}
public void stop()
{
System.out.println("this is stop() method");
```

```
/*<applet code="HelloJava.class"
        width=300
        height=300>
</applet>*/
```

```
}
public void destroy()
{
System.out.println("this is destroy() method");
}
public void paint (Graphics g)
{
try{
int x=Integer.parseInt(getParameter("a"));
int y=Integer.parseInt(getParameter("b"));
g.drawString ("Sum="+(x+y),10,100);
System.out.println("this is paint() method");
}
catch(Exception e)
{
System.out.println("Give input Value in interger type");
}
}
}
```