

Java I/O

- **Java I/O** (Input and Output) is used to process the input and produce the output.
- The java.io package contains all the classes required for input and output operations.
- Java uses the concept of stream to make I/O operation fast.

Stream

Java programs perform I/O through streams.

A stream is a sequence of data or a channel through which data flows from one point to another point. In Java a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

A stream is a method to sequentially access a file. A stream is an abstraction that either produces or consumes information.

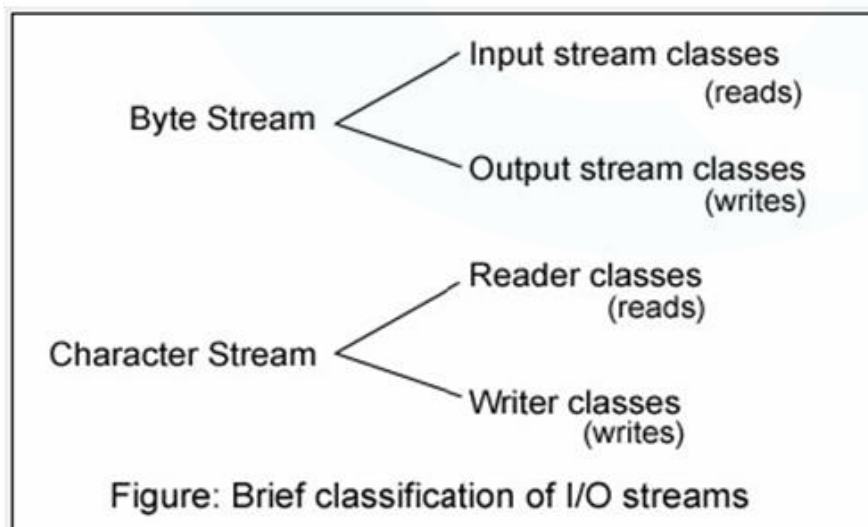
A stream is linked to a physical device by the Java I/O system.

All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection.

Java defines two types of streams: **byte** and **character**.

- **Byte streams:** Java byte streams are used to perform input and output of 8-bit bytes.
- **Character streams:** Java Character streams are used to perform input and output for 16-bit unicode.

Classification of Java Stream classes



Byte Streams

ANUSREE K ,CSE DEPT.

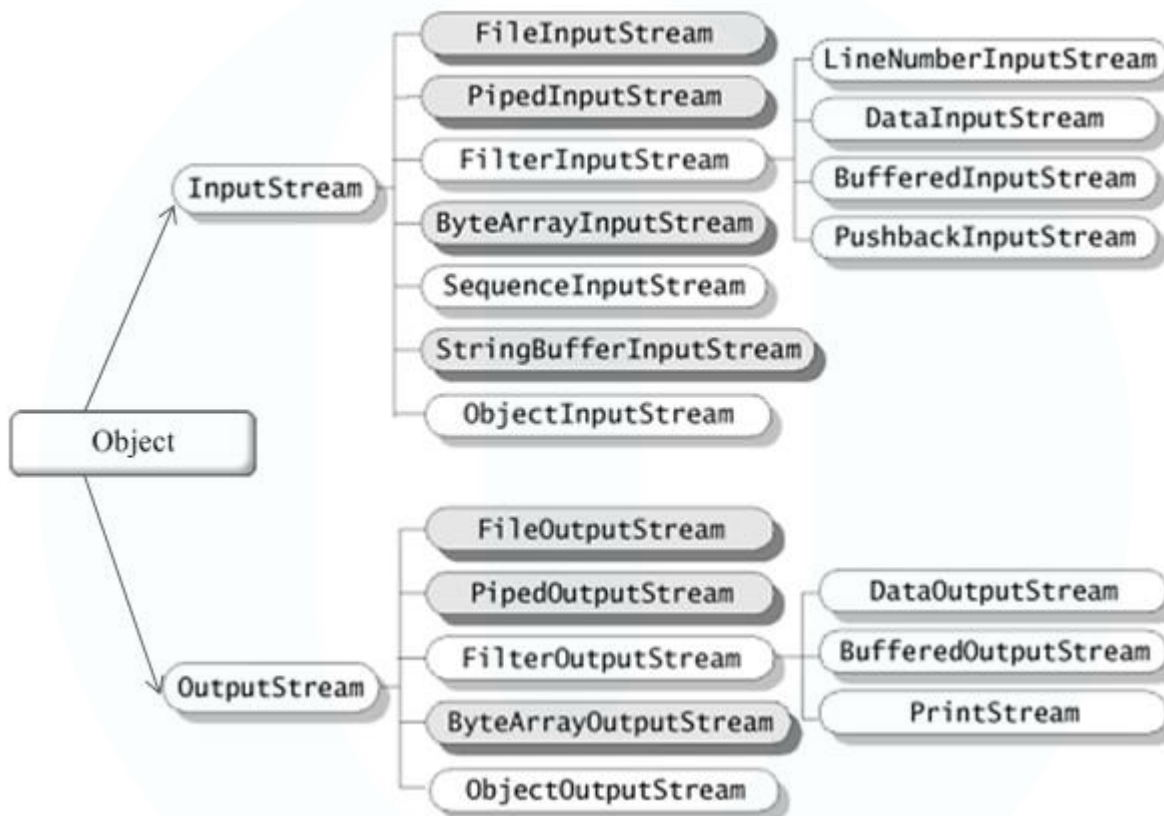
anusreek@sahrdaya.ac.in

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**.

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.

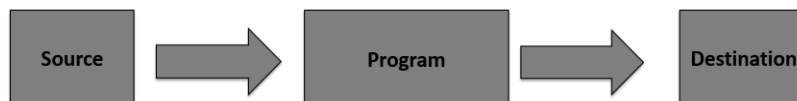
To use the stream classes, we must import **java.io**.

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



OutputStream vs InputStream

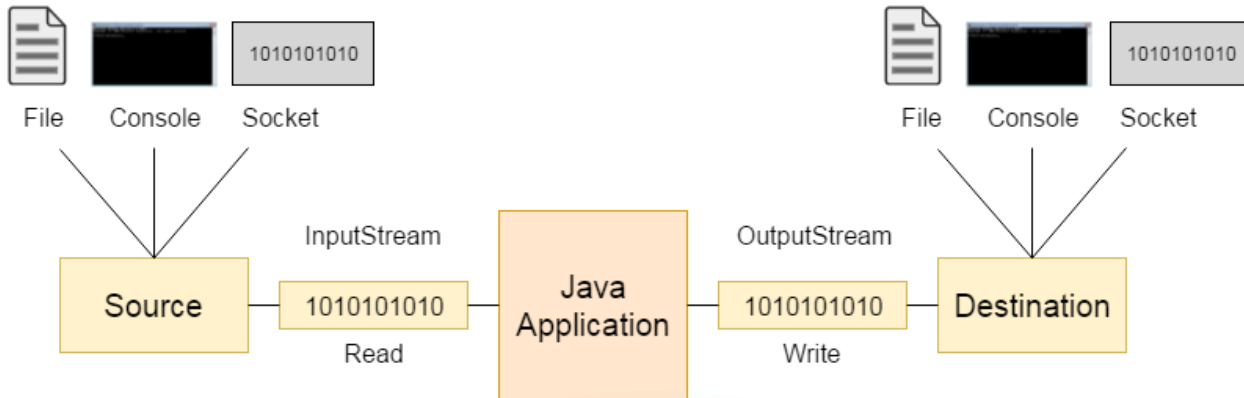
- **InPutStream:** The InputStream is used to read data from a source, it may be a file, an array, peripheral device or socket.
- **OutPutStream:** the OutputStream is used for writing data to a destination, it may be a file, an array, peripheral device or socket.



Working of Java OutputStream and InputStream

ANUSREE K ,CSE DEPT.

anusreek@sahrdaya.ac.in



InputStream class

InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) read()	Reads the next byte of data from the input stream. It returns -1 at the end of file.
2) available()	Gives number of bytes available in the input stream.
3) close()	Closes the current input stream.

OutputStream class

OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) write(int)	Used to write a byte to the current output stream.
2) write(byte[])	Used to write an array of byte to the current output stream.
3) flush()	Flushes the current output stream.
4) close()	Used to close the current output stream.

Following is an example which makes use of `FileInputStream` and `FileOutputStream` classes to copy an input file into an output file:

```
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();
            }
        }
    }
}
```

Create a file **input.txt** with the following content:

This is test for copy file.

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

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

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**.

ANUSREE K ,CSE DEPT.

anusreek@sahrdaya.ac.in

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

The 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.

We can re-write above example which makes use of FileReader and FileWriter classes to copy an input file (having unicode characters) into an output file:

```
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();
            }
        }
    }
}
```

Create a file **input.txt** with the following content:

This is test for copy file.

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

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen.

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

Java provides 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 to 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 to standard error stream and represented as **System.err**.

Code to print **output and error** message to the console.

1. `System.out.println("simple message");`
2. `System.err.println("error message");`

Code to get **input** from console.

3. `int i=System.in.read();//returns ASCII code of 1st character`
4. `System.out.println((char)i);//will print the character`

Reading Console Input

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

Java BufferedReader Class:

- Java **BufferedReader** class is used to read the text from a character-based input stream. It can be used to read data line by line by `readLine()` method. It makes the performance fast. It inherits **Reader** class.

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

BufferedReader(Reader inputReader)

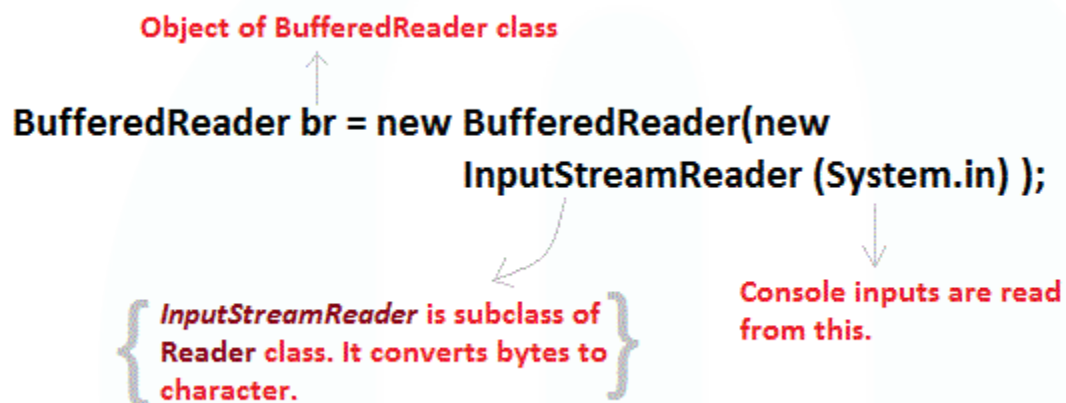
Here, `inputReader` is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**,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

InputStreamReader(InputStream inputStream)

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

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

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



Reading Characters

To read a character from a **BufferedReader**, use **read()**. The version of **read()** that we will be using is `int read()` throws `IOException`. Each time that **read()** is called, it reads a character from the input stream and returns it as an integer value. It returns `-1` when the end of the stream is encountered.

Reading Strings

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

`String readLine()` throws `IOException`

// Use a BufferedReader to read characters from the console.

```
import java.io.*;
class BRRead
{
    public static void main(String args[]) throws IOException
    {
        char c;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```

        System.out.println("Enter characters, 'q' to quit.");
        do
        {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}

```

Output:

Enter characters, 'q' to quit.

123abcq

1

2

3

a

b

c

q

Write a program to find addition of two numbers given from the keyboard. Using exception handling, display appropriate message if the inputs are not valid numbers.

```

import java.io.*;

class Addition
{
    public static void main(String args[])
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        int x, y, z;
        try
        {
            System.out.println("Enter 1st number");
            str = br.readLine();
            x=Integer.parseInt(str);
            System.out.println("Enter 2nd number");
            str = br.readLine();
            y=Integer.parseInt(str);
            z=x+y;
            System.out.print("Addition Result is: "+z);
        }
        catch (Exception e)
        {
            System.out.println("Invalid Number ..Try Again!!!!!!");
        }
    }
}

```



```
}
```

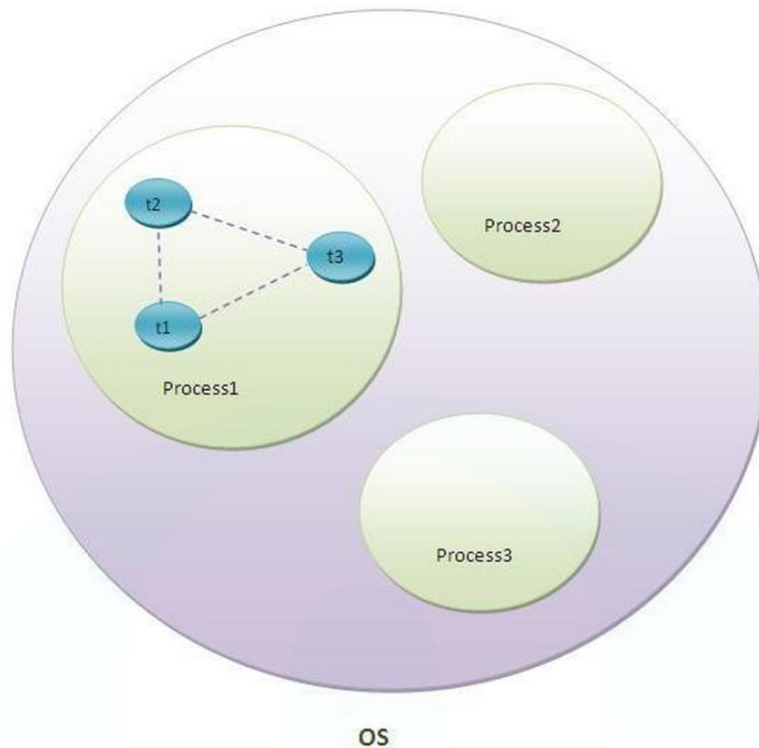
Write a program to find factorial of an integer given from the keyboard. Using exception handling mechanism, display appropriate message if the input from keyboard is not a valid integer.

```
import java.io.*;

class Factorial
{
    public static void main(String args[])
    {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        int n, f=1, i;
        try
        {
            System.out.println("Enter the number");
            str = br.readLine();
            n=Integer.parseInt(str);
            for(i=1; i<=n; i++)
                f=f*i;
            System.out.print("Factorial is: "+f);
        }
        catch (Exception e)
        {
            System.out.println("Invalid Input ..Try Again!!!!!!");
        }
    }
}
```

Thread in java

- A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



Thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Difference between Process and Thread in Java

Process	Thread
An executing program is called a process.	A thread is a small part of a process.
Every process has its separate address space.	All the threads of a process share the same address space cooperatively as that of a process.
Process-based multitasking allows a computer to run two or more than two programs concurrently.	Thread-based multitasking allows a single program to run two or more threads concurrently.
Communication between two processes is expensive and limited.	Communication between two threads is less expensive as compared to process.
Context switching from one process to another process is expensive.	Context switching from one thread to another thread is less expensive as compared to process.
Process are also called heavyweight task.	Thread are also called lightweight task.

Life cycle of a Thread (Thread States)

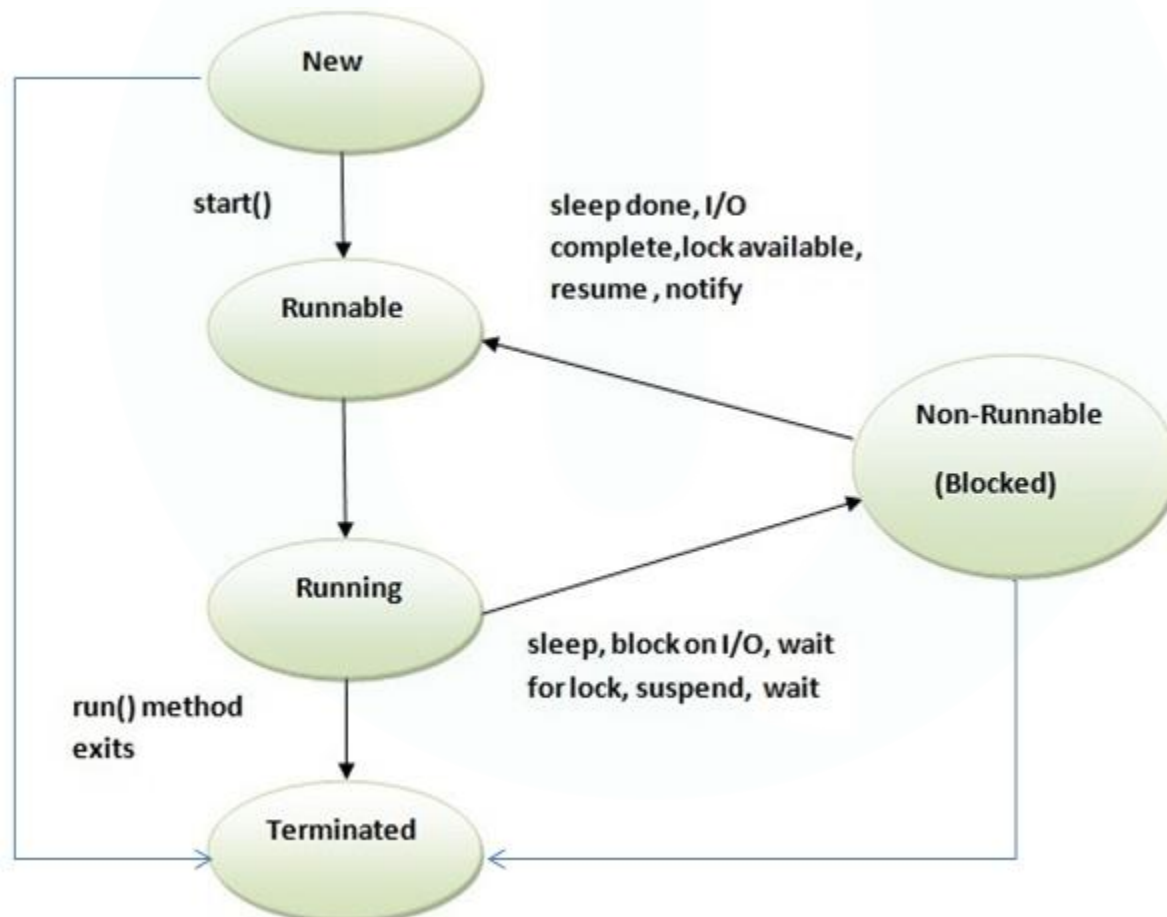
A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies.

A thread can be in one of the five states. The life cycle of the thread in java is controlled by JVM.

Java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

The Following diagram shows complete life cycle of a thread.



- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

- **Blocked (Waiting):** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead):** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Creating Thread by Extending Thread class

First way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

Thread class provide constructors and methods to create and perform operations on a thread.

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.

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create Thread class object.

Extending the Thread Class:

1. Declare the class as extending the Thread class.
2. Implement the run() method that is responsible for executing the sequence of code that thread will execute.

public void run(): is used to perform action for a thread.

3. Create a thread object and call the start() method to initiate the thread execution

void start();

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
}
}
```

Output: thread is running...

Creating Thread by Implementing Runnable interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface. To implement Runnable, a class need only implement a single method called **run()**.

Runnable interface have only one method named run().

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
}
}
```

Output: thread is running...

Commonly used methods of Thread class:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.

- **public void sleep(long milliseconds):** blocked for a specified time.
- **public void suspend():** blocked until further orders.
- **public void stop():** is used to stop the thread.
- **public void resume():** is used to resume the suspended thread.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void wait():** blocked until certain condition occurs.

Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

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.

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it.

- A monitor is an object that is used as a mutually exclusive lock, or mutex. The thread that holds the key can only open the lock and only one thread can own a monitor at a given time.
- A thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
- 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.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

You can synchronize your code in either of two ways. Both involve the use of the synchronized keyword.

1. by synchronized method
2. by synchronized block

1. Using Synchronized Methods:

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

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.

Syntax to use synchronized Method

```
synchronized Return_type
function_name()
{
    //code here is synchronized
}
```

Example:

```
synchronized void run()
{
    //Synchronized block
}
```

Thread Synchronization using synchronized method

//example of java synchronized method

```
class Table
{
    synchronized void printTable(int n){//synchronized method
    for(int i=1;i<=5;i++){
        System.out.println(n*i);
    try{
        Thread.sleep(400);}
    catch(Exception e){System.out.println(e);}
    } }
}
```

```
class MyThread1 extends Thread
{
```

```
Table t;
MyThread1(Table t){
    this.t=t; }
    public void run(){
        t.printTable(5);}
}
class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t){
        this.t=t;}
    public void run(){
        t.printTable(100); }
}
public class TestSynchronization2
{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start(); }
}
```

Output: 5

10
15
20
25
100
200
300
400
500

Synchronized block in java (Using synchronized Statement)

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- Synchronized block can be used to perform synchronization on any specific resource of the method.

- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Syntax to use synchronized block

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

Example:

```
public void run()
{
synchronized(fobj)      //Synchronized
block
{
        fobj.display(msg);
}
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Thread Synchronization using synchronized statement.

```
class Table
{
    void printTable(int n){
        synchronized(this){//synchronized block
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        } } //end of the method
    }
}
class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t){
        this.t=t; }
    public void run(){
        t.printTable(5); }
}
class MyThread2 extends Thread
```

```

{
    Table t;
    MyThread2(Table t){
        this.t=t; }
    public void run(){
        t.printTable(100); }
}

public class TestSynchronizedBlock1
{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start(); }
}

```

Output: 5

```

10
15
20
25
100
200
300
400
500

```

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- 2) You **can perform many operations together so it saves time**.
- 3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- A **process-based** multitasking is the feature that allows our computer to run two or more programs concurrently. For example, process-based multitasking enables us to run the Java compiler at the same time that we are using a music player.
- Each process has its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- In a **thread-based** multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

Multiple Thread Creation by implementing Runnable interface

```
class RunnableInterface implements Runnable  
{
```

```
    Thread t;
```


```
ANUSREE K ,CSE DEPT.
```

anusreek@sahrdaya.ac.in

```

int i;
RunnableInterface()
{
    t=new Thread(this,"Child Thread");
    System.out.println("Child Thread is: "+t);
    t.start();
}
public void run()
{
    try
    {
        for(i=1;i<=5;i++){
            System.out.println("Child Thread:"+i);
            t.sleep(500);}
        }
    catch(InterruptedException e) {
        System.out.println("Child Thread is Interrupted");}
}
}
class MultipleThreadsRunnableInterface
{
    public static void main(String args[])
    {
        new RunnableInterface();
        try
        {
            for(int i=1;i<=5;i++) {
                System.out.println("Main Thread :"+i);
                Thread.sleep(1000); }
            }
        catch(InterruptedException e) {
            System.out.println("Main Thread Interrupted"); }
    }
}

```

Output:


```

F:\Java>javac MultipleThreadsRunnableInterface.java
F:\Java>java MultipleThreadsRunnableInterface
Child Thread is: Thread[Child Thread,5,main]
Main Thread :1
Child Thread:1
Child Thread:2
Child Thread:3
Main Thread :2
Child Thread:4
Main Thread :3
Child Thread:5
Main Thread :4
Main Thread :5
F:\Java>_

```

Multiple Thread Creation by extending Thread class

```
class A extends thread
{
    public void run()
    {
        for (int i=1;i<=5;i++)
        {
            System.out.println("\tFrom ThreadA:i="+i);
        }
        System.out.println("Exit from A");
    }
}
class B extends Thread
{
    public void run()
    {
        for (int i=1;i<=5;i++)
        {
            System.out.println("\tFrom ThreadB:i="+i);
        }
        System.out.println("Exit from B");
    }
}
class C extends Thread
{
    public void run()
    {
        for (int i=1;i<=5;i++)
        {
            System.out.println("\tFrom ThreadC:i="+i);
        }
        System.out.println("Exit from C");
    }
}
class ThreadTest
{
    public static void main(String args{ })
    {
        new A().start();
        new B().start();
        new C().start();
    }
}
```

Output:*First run*

From	Thread	A	:	i	=	1
From	Thread	A	:	i	=	2
From	Thread	B	:	j	=	1
From	Thread	B	:	j	=	2
From	Thread	C	:	k	=	1
From	Thread	C	:	k	=	2
From	Thread	A	:	i	=	3
From	Thread	A	:	i	=	4
From	Thread	B	:	j	=	3
From	Thread	B	:	j	=	4
From	Thread	C	:	k	=	3
From	Thread	C	:	k	=	4
From	Thread	A	:	i	=	5
Exit from A						
From	Thread	B	:	j	=	5
Exit from B						
From	Thread	C	:	k	=	5
Exit from C						

Second run

From	Thread	A	:	i	=	1
From	Thread	A	:	i	=	2
From	Thread	C	:	k	=	1
From	Thread	C	:	k	=	2
From	Thread	A	:	i	=	3
From	Thread	A	:	i	=	4
From	Thread	B	:	j	=	1
From	Thread	B	:	j	=	2
From	Thread	C	:	k	=	3
From	Thread	C	:	k	=	4
From	Thread	A	:	i	=	5
Exit from A						
From	Thread	B	:	k	=	4
From	Thread	B	:	j	=	5
From	Thread	C	:	k	=	5
Exit from C						
From	Thread	B	:	j	=	5
Exit from B						