UNIT-4

Exception handling and Multithreading

Syllabus:

Exceptions - Introduction, Exception handling techniques-try...catch, throw, throws, finally block, user defined exception.

Multithreading: Introduction, Differences between multi threading and multitasking, The main Thread, Creation of new threads, Thread priority, Multithreading- Using isAlive() and join(), Synchronization, suspending and Resuming threads, Communication between Threads

Exception handling:

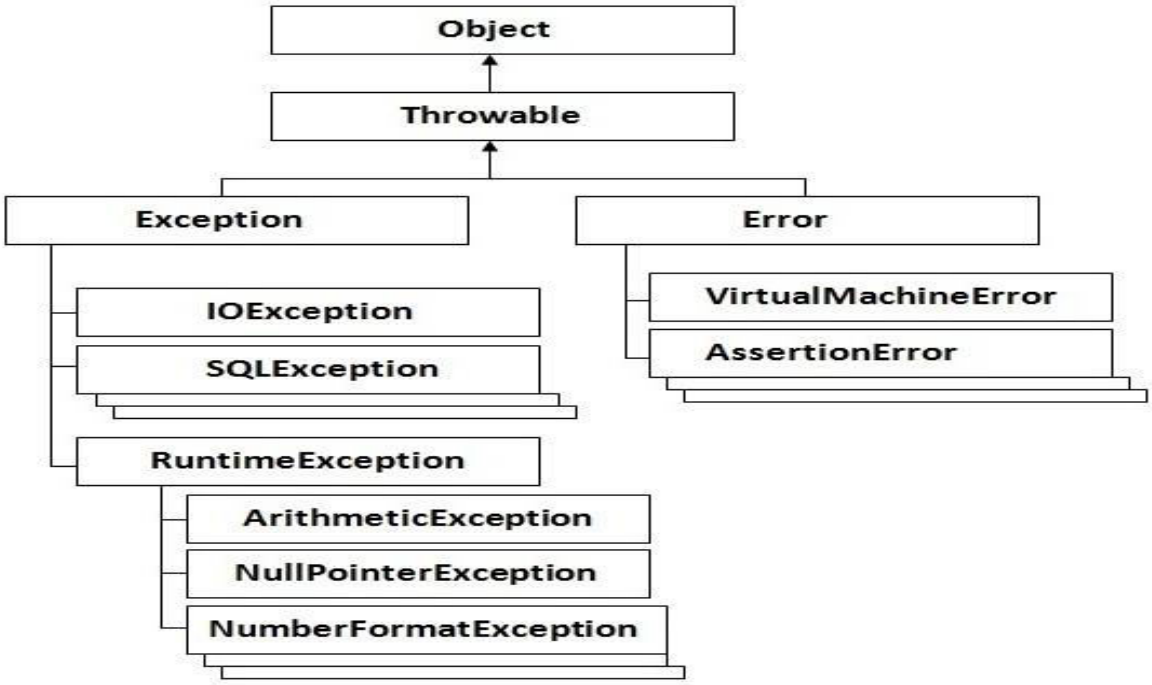
- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- **Exception:** In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- Generally exception is an abnormal condition.
- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.
- The core advantage of exception handling is **to maintain the normal flow of the application**.
- Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

- Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

`

Hierarchy of Java Exception classes



Types of Exception:

- There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception.
- The sun microsystem says there are three types of exceptions:
    1. Checked Exception
    2. Unchecked Exception
    3. Error

Checked Exception:

- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions
  Examples: IOException, SQLException etc.
- Checked exceptions are checked at compile-time.

Unchecked Exception:

- The classes that extend RuntimeException are known as unchecked exceptions
  Examples: ArithmeticException, NullPointerException,
  ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

Error:

- Error is irrecoverable
  Examples: OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur:

- There are given some scenarios where unchecked exceptions can occur as follows:

Where ArithmeticException occurs?

- If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

Where NullPointerException occurs?

- If we have null value in any variable, performing any operation by the variable occurs an NullPointerException

```
String s=null;
System.out.println(s.length());//NullPointerException
```

Where NumberFormatException occurs?

- The wrong formatting of any value, may occur NumberFormatException.
- Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

Where ArrayIndexOutOfBoundsException occurs?

- If we are given any value in the wrong index, result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

try-catch block:

- try block is used to enclose the code that might throw an exception. It must be used within the method.
- catch block is used to handle the Exception. It must be used after the try block only.
- We can use multiple catch block with a single try and try block must be followed by either catch or finally block.
- **Syntax of try-catch**

```
try
{
      //code that may throw exception
}
catch(Exception_class_Name ref)
{
}
```

- **Problem without try-catch**

```
class Testtrycatch
{
      public static void main(String args[])
      {
                  int data=50/0;//may throw exception
                  System.out.println("rest of the code...");
       }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

  - In the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).
  - There can be 100 lines of code after exception. So all the code after exception will not be executed.

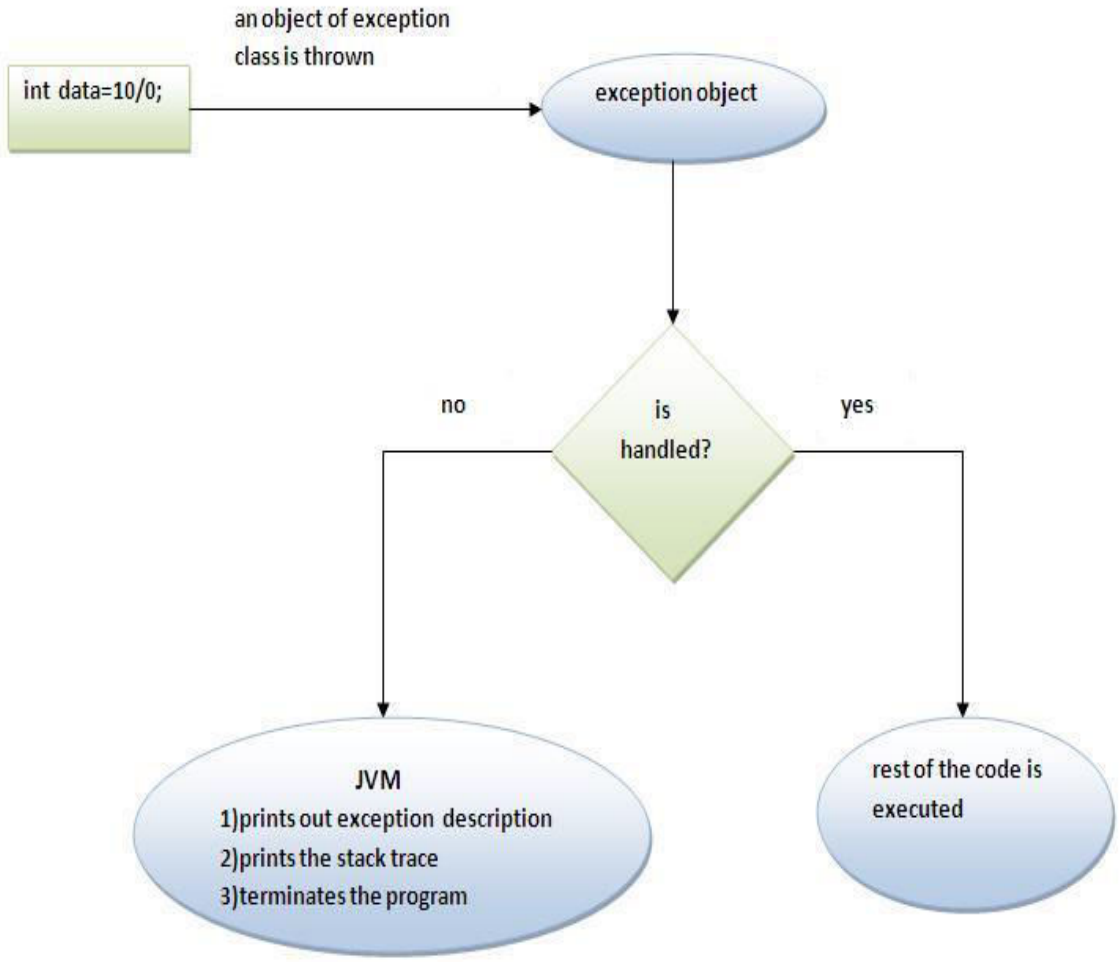
- **Solution by try-catch**

```
class Testtrycatch
{
       public static void main(String args[])
      {
            try
            {
                        int data=50/0;
            }
             catch(ArithmeticException e)
            {
                  System.out.println(e);
            }
            System.out.println("rest of the code...");
      }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

- Now, as displayed in the example, rest of the code is executed i.e. rest of the code... statement is printed.
- **Internal working of java try-catch block**



- The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:
    - Prints out exception description.
    - Prints the stack trace (Hierarchy of methods where the exception occurred).
    - Causes the program to terminate.
- But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

****

Nested try block:

- The try block within a try block is known as nested try block.
- **Why we use nested try block?**
  Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.
- **Syntax:**

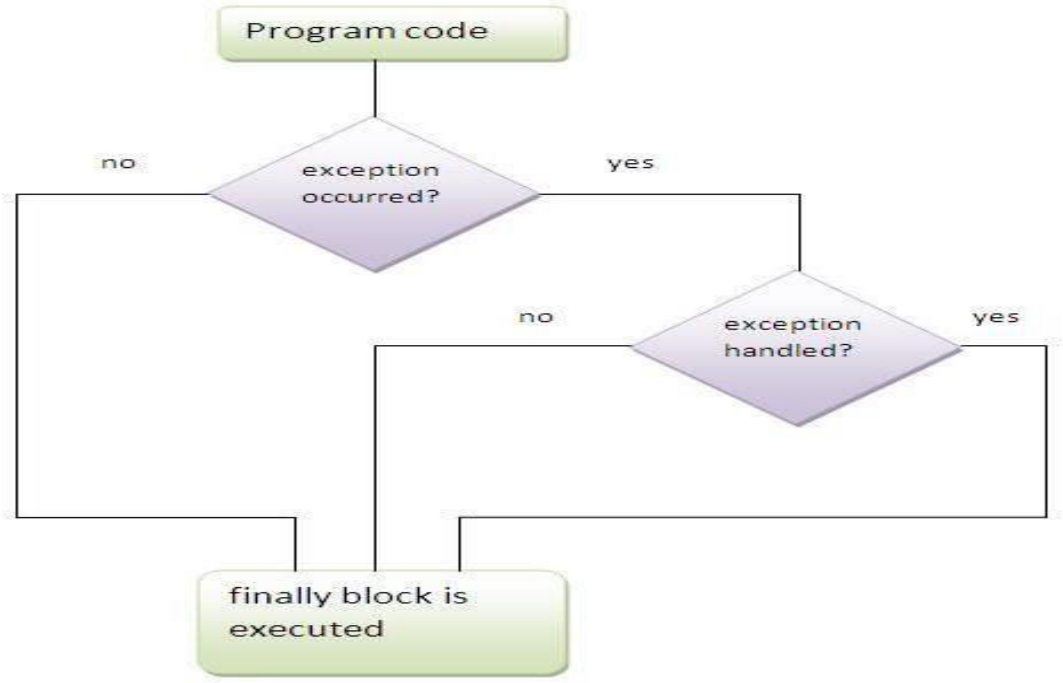
```
try
{
    // statements
    try
    {
        // statements
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
```

- **Example:**

```
class NestedTry
{
   public static void main(String args[])
   {
      try
        {
            int a=args.length;
            int data=50/a;
            System.out.println("a= "+a);
            try
            {
                  if(a==1)
                  {
                      int[] c={1};
                      c[30]=101;
                  }
            }
            catch(ArrayIndexOutOfBoundsException e1)
            {
                  System.out.println(e1);
            }
        }
        catch(ArithmeticException e2)
        {
                  System.out.println(e2);
        }
   }
}
```

finally block:

- **finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- finally block is always executed whether exception is handled or not.
- finally block follows try or catch block.



- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of finally block:

- **Case 1: Example where exception doesn't occur.**

```
class TestFinallyBlock
{
  public static void main(String args[])
  {
     try
     {
       int data=25/5;
       System.out.println(data);
     }
     catch(Exception e)
     {
       System.out.println(e);
     }
```

```
            finally
            {
                System.out.println("finally block is always
                                                    executed");
            }
            System.out.println("rest of the code...");
        }
    }
```

Output: 5
finally block is always executed
rest of the code...

- **Case 2: Example where exception occurs and not handled.**

```
class TestFinallyBlock
{
   public static void main(String args[])
   {
      try
      {
         int data=25/0;
         System.out.println(data);
      }
      catch(NullPointerException e)
      {
         System.out.println(e);
      }
      finally
      {
         System.out.println("finally block is always
                                                executed");
      }
      System.out.println("rest of the code...");
   }
}
```

Output:
finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

- **Case 3: example where exception occurs and handled.**

```
class TestFinallyBlock
{
   public static void main(String args[])
   {
      try
      {
         int data=25/0;
         System.out.println(data);
      }
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      finally
      {
         System.out.println("finally block is always
                                                executed");
      }
      System.out.println("rest of the code...");
   }
}
```

Output:
```
Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...
```

****

throw keyword:

- The throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.
- The syntax of throw keyword is given below.

```
throw new Exception_Name("message”);
```

- **Example:** In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
class TestThrow
{
   static void validate(int age)
   {
      if(age<18)
            throw new ArithmeticException("not valid");

      else
           System.out.println("welcome to vote");
   }
   public static void main(String args[])
   {
      validate(13);
      System.out.println("rest of the code...");
   }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

****

throws keyword:

- The **throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions.
- If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.
- **Syntax of throws**

```
returntype methodname() throws exception_class_name
{
        //method code
}
```

- **Example-1:**

```
import java.io.*;
class Testthrows
 {
    void m()throws IOException
    {
        throw new IOException("device error");//checked exception
    }
    void p()
    {
       try
       {
          m();
       }
       catch(Exception e)
       {
          System.out.println("exception handled");
       }
    }
    public static void main(String args[])
    {
         Testthrows obj=new Testthrows();
         obj.p();
         System.out.println("normal flow...");
    }
}
```

Output:

```
exception handled
normal flow...
```

- **Example-2:**

```
import java.util.*;
class ThrowsExceptionClass
{
public static void main(String args[])throws InputMismatchException
        {
             int a,b;
            Scanner s=new Scanner(System.in);
            System.out.println("Enter a, b values");
            try
            {
                 a=s.nextInt();
                 b=s.nextInt();
            }
            catch(InputMismatchException e)
            {
                System.out.println(e);
            }
        }
    }
```

Output:

```
Enter a, b values
12
13.45
java.util.InputMismatchException
```

****

User defined Exception (or) Custom Exception:

- If we are creating our own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, we can have our own exception and message.
- If we want to create any user defined exception, that must extends "Exception" class '
- **General form is**

```
class CustomExceptionName extends Exception
{
  CustomExceptionName(String s)
  {
    super(s);//calling constructor of Exception class
  }
}
```

- **Example**

```
class OwnException extends Exception
{
  OwnException(String s)
  {
        super(s);
  }
}
class OwnExceptionDemo
{
   static void validate(int age)throws OwnException
   {
       if(age<18)
        {
            throw new OwnException("age is not valid");
        }
        else
        {
            System.out.println("Welcome to vote");
        }
   }
   public static void main(String args[])
   {
       try
        {
           validate(13);
        }
        catch(OwnException e)
        {
           System.out.println(e);
        }
   }
}
```

Output:

```
OwnException: age is not valid
```

****

Difference between throw and throws:

Throw	throws
throw keyword is used to explicitly throw an exception.	throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
throw is followed by an instance.	throws is followed by class.
throw is used within the method.	throws is used with the method signature.
We cannot throw multiple exceptions.	We can declare multiple exceptions Ex: `public void method()throws IOException,SQLException`

Example:
`void m()`
`{`
`throw new ArithmeticException("sorry");`
`}`
	Example:
	`void m()throws ArithmeticException`
	`{`
	`//method code`
	`}`

Multithreading:

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

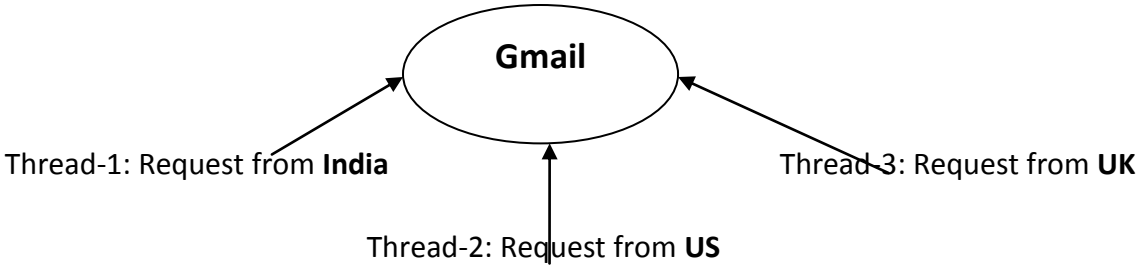
- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:
    - Process-based Multitasking(Multi processing)
    - Thread-based Multitasking(Multi threading)

➢ **Process-based Multitasking (Multiprocessing)**

- Process of executing multiple tasks simultaneously where each task is a individual process.
- This type of multitasking mostly used in Operating System level.
- Example: Typing program in editor, listening audio songs and downloading files from internet simultaneously.

➢ **Thread-based Multitasking (Multithreading)**

- Executing several tasks simultaneously or parallely where each task is a individual process in a single program or application.
- Here each process is independent to each other there is no dependency.
- Each individual process is called as Thread.
- Example: Gmail Server



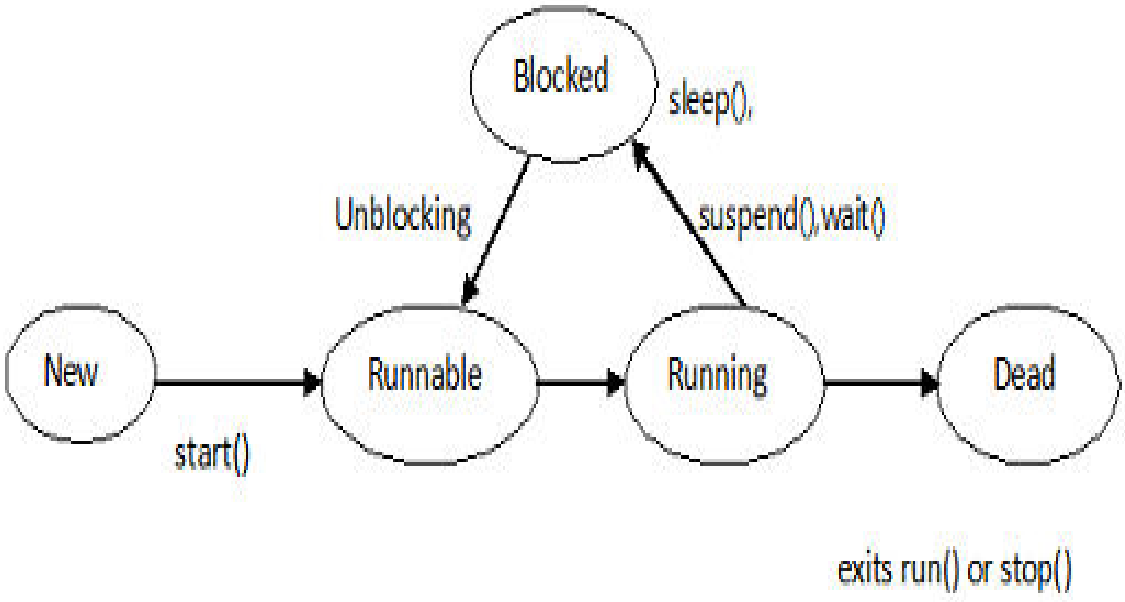
Thread:

- 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 Life cycle:

- A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
- But for better understanding the threads, we are explaining it in the 5 states.
- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New (or) Born
2. Runnable (or) Ready
3. Running
4. Blocked (or) Non-Runnable
5. Dead (or) Terminated



1. **Newborn State:** When a thread object is created a new thread is born and said to be in Newborn state.
2. **Runnable State:** If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion

3. **Running State:** It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs
    a. Thread give up its control on its own and it can happen in the following situations
        i. A thread gets suspended using **suspend()** method which can only be revived with **resume()** method
        ii. A thread is made to sleep for a specified period of time using s**leep(time)** method, where time in milliseconds
        iii. A thread is made to wait for some event to occur using **wait ()** method. In this case a thread can be scheduled to run again using **notify ()** method.
    b. A thread is pre-empted by a higher priority thread
4. **Blocked State:** If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.
5. **Dead State:** A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.

****

The main Thread:

- When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of our program, because it is the one that is executed when our program begins.
- Although the main thread is created automatically when our program is started, it can be controlled through a **Thread** object.
- To do so, we 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 we have a reference to the main thread, we can control it just like any other thread.

```
class CurrentThreadDemo
{
   public static void main(String args[])
   {
        Thread t = Thread.currentThread();
         // getting name of thread
        System.out.println("Current thread: " + t.getName());
        // change the name of the thread
       t.setName("My Thread");
        System.out.println("name of Thread:" + t.getName());
```

```
        for(int n = 5; n > 0; n--)
        {
            System.out.println(n);
        }
    }
}
```

Output:

```
Current thread: main
name of Thread:My Thread
5
4
3
2
1
```

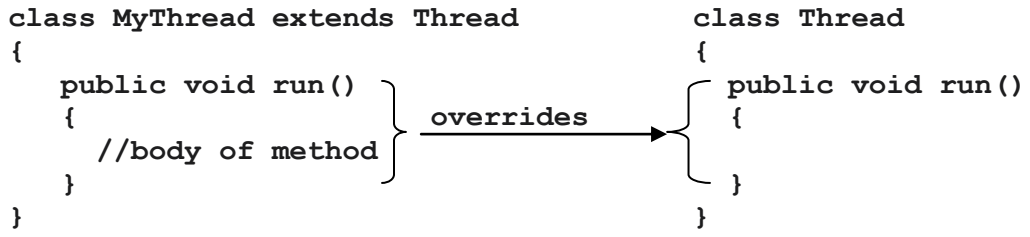
****

Creation of new threads:

- For creating threads we have two ways
    1. By extending **Thread** class
    2. By implementing **Runnable** interface

Creating a thread by extending Thread class:

- For creating 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.
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread.



- **Example:**

```
class MyThread extends Thread
{
   public void run()
   {
       for(int i=1;i<=4;i++)
         {
            System.out.println("MyThread: "+i);
         }
   }
}
```

```
class ThreadCreationDemo
{
   public static void main(String s[])
   {
          MyThread t=new MyThread();
          t.start();
   }

}
```

Output:

```
MyThread: 1
MyThread: 2
MyThread: 3
MyThread: 4
```

Creating a thread by implementing Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.

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

implements

```
interface Runnable
{
   public void run();
}
```

- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.

```
Thread t=new Thraed(obj);
```

Here 'obj' is the instance of class that implements Runnable interface.

- **Example:**

```
class MyThread implements Runnable
{
   public void run()
   {
       for(int i=1;i<=4;i++)
        {
           System.out.println("MyThread: "+i);
        }
   }
}
class ThreadCreationDemo
{
   public static void main(String s[])
   {
          MyThread mt=new MyThread();
          Thread t=new Thread(mt);
```

```
            t.start();
    }

}
```

Output:

```
MyThread: 1
MyThread: 2
MyThread: 3
MyThread: 4
```

Creating multiple threads:

```
class MyThread1 extends Thread
{
   public void run()
   {
       for(int i=1;i<=4;i++)
         {
            System.out.println("MyThread1: "+i);
         }
   }
}
class MyThread2 extends Thread
{
   public void run()
   {
       for(int i=1;i<=4;i++)
         {
            System.out.println("MyThread2: "+i);
         }
   }
}
class MultiThreadDemo
{
   public static void main(String s[])
   {
          MyThread1 t1=new MyThread1();
          MyThread2 t2=new MyThread2();
          t1.start();
          t2.start();
   }
}
```

Output:

Output may be changed from run to run. Because both threads executes simultaneously.

Getting and setting names of Thread:

- For getting name of thread we use following method

```
public final String getName()
```

- For getting name for a thread we use following method

```
public final void setName(String name)
```

- **Example**

```
class MyThread1 extends Thread
{
   public void run()
   {
   }
}
class GettingSettingName
{
   public static void main(String s[])
   {
      MyThread1 t1=new MyThread1();
      System.out.println(Thread.currentThread().getName());
      Thread.currentThread().setName("my main");
      System.out.println(Thread.currentThread().getName());
      t1.start();
      System.out.println(t1.getName());
      t1.setName("my thread");
      System.out.println(t1.getName());
   }
}
```

Output:

```
main
my main
Thread0
my thread
```

****

Thread Priorities:

- In java every thread has some priority. Based on priority thread scheduler allocates processor for each thread. The default priority of main thread is 5
- The thread priorities are within a range of 1 to 10.
- If we create any thread, then the priority of thread is main thread priority i.e. 5 , because the default parent of Thread is main thread.
- We have 3 types of priorities
    1. Minimum priority: 1
    2. Normal priority: 5
    3. Maximum priority: 10
- To display priorities of thread the following code is suitable

```
class ThreadPriorities
{
      public static void main(String args[])
      {
            System.out.println(Thread.MIN_PRIORITY); //1
            System.out.println(Thread.NORM_PRIORITY); //5
            System.out.println(Thread.MAX_PRIORITY); //10
      }
}
```

Getting and setting priorities of Thread:

- For getting name of thread we use following method

```
      public final int getPriority ()
```

- For getting name for a thread we use following method

```
     public final void setPriority (int value)
```

- **Example**

```
class MyThread1 extends Thread
{
   public void run()
   {
   }
}
class GettingSettingName
{
   public static void main(String s[])
   {
       MyThread1 t1=new MyThread1();
       System.out.println(Thread.currentThread().getPriority());
       System.out.println(t1.getPriority ());
       t1.setPriority (8);
       System.out.println(t1.getPriority ());
   }

}
```

Output:

```
5
5
8
```

isAlive():

- This method is returns status of a thread i.e. returns Boolean values either true or false.
- The general form of method is

```
public final Boolean isAlive()
```

- **Example :**

```
class MyThread extends Thread
{
   public void run()
   {
   }
}
class ThreadCreationDemo
{
   public static void main(String s[])
   {
          MyThread t=new MyThread();
          System.out.println(Thread.currentThread().isAlive());
          t.start();
          System.out.println(t.isAlive());
   }
}
```

Output:

```
true
true
```

sleep():

- The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

- The number of milliseconds to suspend is specified in *milliseconds.* This method may throw an **InterruptedException**.
- The **sleep( )** method has a second form, which allows you to specify the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds)
                                     throws InterruptedException
```

- **Example:**

```
class MyThread1 extends Thread
{
    public void run()
      {
          for(int i=1;i<=4;i++)
            {
                  try
                  {
                     Thread.sleep(1000); // sleeps for 1 second
                     System.out.println("Child Thread "+i);
                  }
                  catch(InterruptedException e)
                  {
                  }
            }
      }
}

class ThreadSleepDemo
{
   public static void main(String args[])throws InterruptedException
      {
         MyThread1 t1=new MyThread1();
          t1.start();
      }
}
```

Output:(for each line printing it takes 1 sec time)

```
Child Thread: 1
Child Thread: 2
Child Thread: 3
Child Thread: 4
```

join():

- **join** method can be used to pause the current thread execution until unless the specified thread is completed. There are three overloaded join methods.

```
public final void join()
public final void join(long millis)
public final void join(long millis, int nanos)
```

- Join method throws InterruptedException. Because when ever join method call then thread execution interrupted for some time.

Example1:

```
class MyThread31 extends Thread
{
     public void run()
       {
           for(int i=1;i<=5;i++)
            {
               try
                 {
                    sleep(2000);
                    System.out.println("Child Thread : "+i);
                 }
                 catch(InterruptedException e){}
            }
       }
}
class ThreadJoinDemo
{
     public static void main(String args[])
       {
           MyThread22 t=new MyThread31();
            t.start();
            try
            {
                 t.join();
            }
            catch(InterruptedException e)
            {
            }
            for(int j=1;j<=5;j++)
            {
                    System.out.println("Main Thread : "+j);
            }
       }
}
```

Output:

```
Child Thread : 1
Child Thread : 2
Child Thread : 3
Child Thread : 4
Child Thread : 5
Main Thread : 1
Main Thread : 2
Main Thread : 3
Main Thread : 4
Main Thread : 5
```

Example2:

```
class MyThread11 extends Thread
{
    public void run()
     {
        String name= Thread.currentThread().getName();
        System.out.println("Thread started: "+name));
        try
         {
            sleep(4000);
        }
         catch (InterruptedException e){ }

        System.out.println("Thread ended:"+name);
    }
}
class ThreadJoinExample
{
    public static void main(String[] args)
      {
        MyThread11 t1 = new MyThread11();
        MyThread11 t2 = new MyThread11();
        t1.start();
        //starts second thread after completing t1 thread
          try
           {
            t1.join();
           }
           catch (InterruptedException e)
           {
           }
           t2.start();
           try
           {
            t2.join();
           }
           catch (InterruptedException e)
           {
           }
   System.out.println("t1,t2 are completed,exiting main thread");
     }
}
```

Output:

```
Thread started: Thread-0
Thread ended:Thread-0
Thread started: Thread-1
Thread ended:Thread-1
t1,t2 are completed,exiting main thread
```

****

Thread Synchronization:

- If multiple threads want to execute simultaneously in some cases we may get data inconsistency problem.
- To overcome data inconsistency problem we use Thread synchronization.
- In java Synchronization can be achieved by using **synchronized** keyword
- **synchronized** keyword is applicable for only methods and blocks but not for classes and variables.
- General form of synchronized method and block.

```
synchronized return_type method_name(arguments)
{
}

synchronized(object)
{
}
```

- In java every object has two areas synchronized area and non-synchronized area.
- In multithreading **at a time only one thread can execute synchronized area**, in this scenario thread acquires the lock of object. Once thread completes execution then it releases lock.
- The released lock is allotted to another thread by Thread Scheduler.
- While executing one thread executing synchronized area the remaining threads are in waiting state until current thread completes its execution.
- The non-synchronized area can execute multiple threads simultaneously.
- **Example:**

```
class Table
{
   synchronized void printTable(int n)
   {
        System.out.println("Multiplication table for :"+n);
        for(int i=1;i<=5;i++)
        {
           System.out.println(n+"*"+i+"="+n*i);
           try
           {
               Thread.sleep(500);
           }
           catch(Exception e)
           {
               System.out.println(e);
           }
        }
       System.out.println("Thread execution completed\n");
   }
}
```

```
class MyThread11 extends Thread
{
     Table t;
     int n;
     MyThread11(Table t,int n)
     {
          this.t=t;
          this.n=n;
     }
     public void run()
     {
          t.printTable(n);
     }
}
class ThreadSynchronization1
{
     public static void main(String args[])
     {
          Table obj = new Table();//only one object
          MyThread11 t1=new MyThread11(obj,10);
          MyThread11 t2=new MyThread11(obj,20);
          t1.start();
          t2.start();
     }
}
```

Output:

```
Multiplication table for :10
10*1=10
10*2=20
10*3=30
10*4=40
10*5=50
Thread execution completed

Multiplication table for :20
20*1=20
20*2=40
20*3=60
20*4=80
20*5=100
Thread execution completed
```

****

Suspending and Resuming and stopping threads:

- For suspending, resuming and stopping threads we use suspend(), resume(), stop() .
- stop(), suspend(), resume() methods are called as deprecated methods. These methods are not recommended to use in multithreading. The general form of these methods

```
final void suspend( )
final void resume( )
void stop()
```

- **Example:**

```
class Thread1 extends Thread
{
    public void run()
     {
        try
        {
            for(int i=1;i<=3;i++)
            {
                 sleep(1000);
                 System.out.println("this is Thread1 :"+i);
            }
        }
        catch(Exception e){}
     }
}
class SuspendResumeDemo
{
   public static void main(String args[])throws Exception
    {
        Thread1 t1=new Thread1();
        t1.start();
        t1.suspend();//its suspends the current thread.
        for(int j=0;j<=3;j++)
        {
            Thread.sleep(1000);
            System.out.println("I am main thread: "+j);
        }
        t1.resume();//its resume the suspended thread.
    }
}
```

Output:

Compilation: `javac SuspendResumeDemo.java`

```
Note: SuspendResumeDemo.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Running: `java SuspendResumeDemo`

```
I am main thread: 1
I am main thread: 2
I am main thread: 3
this is Thread1 :1
this is Thread1 :2
this is Thread1 :3
```

Thread Communication:

- In multithreading thread communication can be achieved using three methods.
  1. **wait()** : waits for some time until it receives notification from thread
  2. **notify()** : gives notification to waiting thread
  3. **notifyAll()** : gives notifications to all waiting threads.
- The general forms

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

- the above methods exist inside "Object" class.
- These methods should be called inside synchronized area i.e. either synchronized method or synchronized block.
- **Example:**

```
class NotifyThread extends Thread
{
    int total;
    public void run()
    {
       System.out.println("NotifyThread starts updation");
       synchronized(this)
       {
          for(int i=1;i<=100;i++)
            total=total+i;
          System.out.println("NotifyThread giving nofication");
          this.notify();
       }
    }
}
class WaitNotifyDemo
{
   public static void main(String s[])throws InterruptedException
    {
       NotifyThread t1=new NotifyThread();
       t1.start();
       synchronized(t1)
        {
          System.out.println("Main thread going to waiting state");
         t1.wait();
        }
        System.out.println("Main thread recieves notification");
        System.out.println("Total is:"+t1.total);
   }
}
```

OutPut:

```
Main thread going to waiting state
NotifyThread starts updation
NotifyThread giving nofication
Main thread recieves notification
Total is:5050
```

****