Unit – IV:

1. Types of errors, exceptions, try...catch statement, multiple catch blocks, throw and throws keywords, finally clause, uses of exceptions, user defined exceptions

Types of errors

Introduction Errors and Exception:

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.
- Error may produce
  - An incorrect output
  - may terminate the execution of program abruptly
  - may cause the system to crash
- It is important to detect and manage properly all the possible error conditions in program.

Types of Errors

1. Compile time Errors: Detected by javac at the compile time
2. Run time Errors: Detected by java at run time
3. Logical Errors: Produces incorrect results.

1. Compile Time Errors(Syntactical errors)

- Errors which are detected by javac at the compilation time of program are known as compile time errors.
- Most of compile time errors are due to typing mistakes, which are detected and displayed by javac.
- Whenever compiler displays an error, it will not create the .class file.
- Typographical errors are hard to find.

The most common problems:

- Missing semicolon
- Missing or mismatch of brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments/ Initialization
- Bad references to objects
- Use of = in place of = = operator etc.
- Other errors are related to directory path like command not found

Example:

```
class ExError
{
        public static void main(String args[])
        {
                System.out.println("Preeti Gajjar") //Missing semicolon
```

```
          }
}
```

Output:

```
Javac detects an error and display it as follow: Error1.java:  5: „;
„expected System.out.println (“Preeti Gajjar”)
^1 error
```

2. **Run time Error(Logical Error)**
   - There is a time when a program may compile successfully and creates a
   - .class file but may not run properly.
   - It may produce wrong results due to wrong logic or may terminate due to
   - errors like stack overflow, such logical errors are known as run time errors.
   - Java typically generates an error message and aborts the program.
   - It is detected by java (interpreter)
   - Run time error causes termination of execution of the program.
   - The most common run time errors are:
     - Dividing an integer by zero.
     - Accessing element that is out of the bounds of an array.
     - Trying a store a value into an array of an incompatible class or type.
     - Passing a parameter that is not in a valid range or value for a method.
     - Attempting to use a negative size of an array
     - Converting invalid string to a number.

Example:

```
 class Error2
{
        public  static void main(String args[])
          {
                 int a=10,b=5 ,c=5;
                 int x = a / (b-c);            // division by zero
                System.out.println(“x=“ + x); int y = a / (b+c);
                System.out.println(“y=“ + y);
          }
}
```

3. **Semantic Errors(Logical Error)**
   - They are known as Logical errors, they occurs when the code runs without errors, but produces incorrect results.
   - Logical errors are caused by mistakes in design or implementation of code.
   - It is difficult to find and fix.

Example:

```
class Error3
{
        public  static void main(String args[])
          {
                 int a=10,b=5
                 int c = a + b;
                     c = a – b;  // Incorrect calculation
```

```
                System.out.println("Result="+c);
        }
  }
```

Output:
Result= 5 instead of 15

Exception Hierarchy

- All exception and error types are subclasses of class **Throwable**, which is the base class of the hierarchy.
- Immediately below Throwable, are two subclasses that partition exceptions into two distinct branches.
    - Exception Class :
        - One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch.
        - Ex.NullPointerException is an example of such an exception.
    - Error Class
        - Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).
        - Ex.StackOverflowError is an example of such an error.

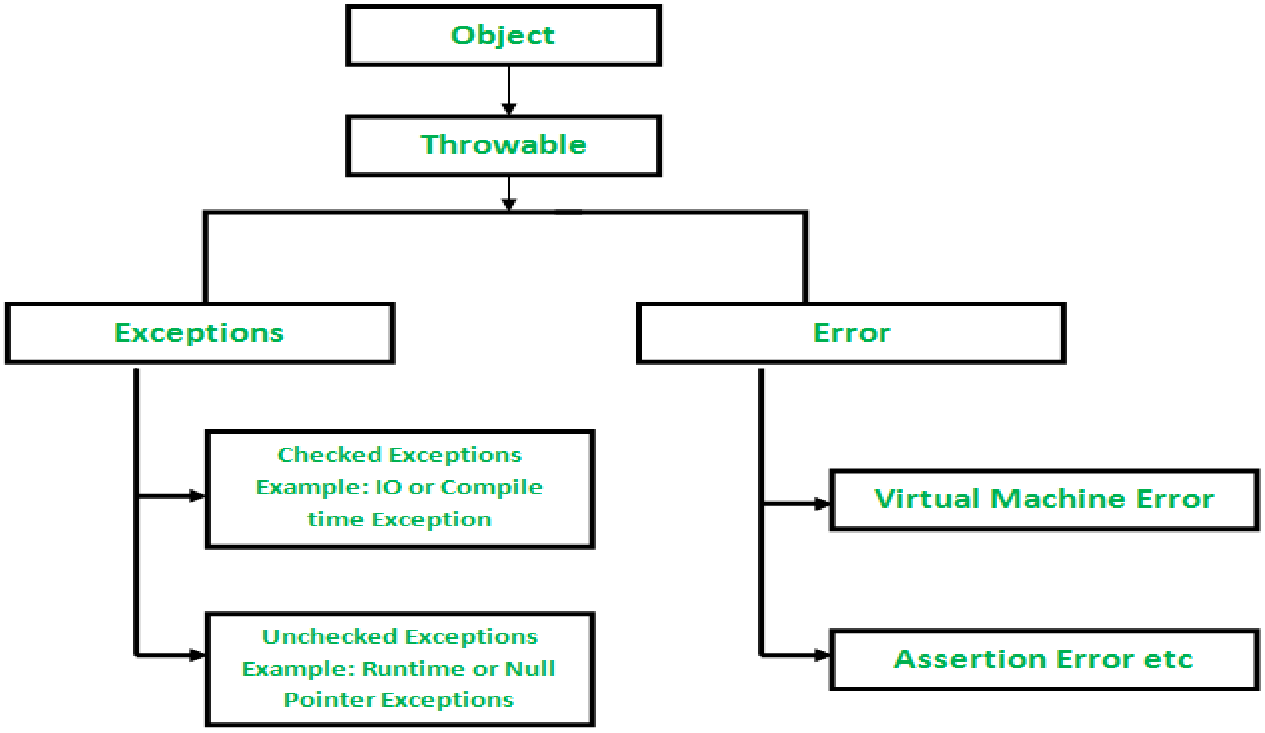
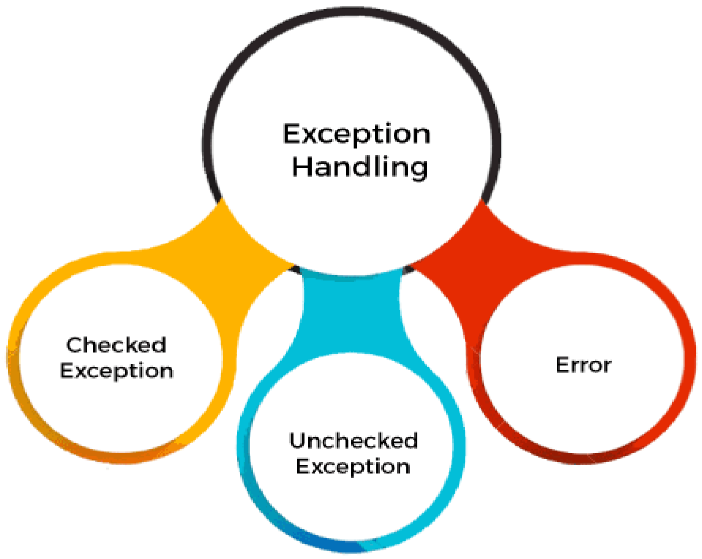

Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Figure: Exception Hierarchy

➢ Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. **Checked Exception**
2. **Unchecked Exception**
3. **Error**



Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

➢ Exception Handling Mechanism:

In Java exception handling is done using **five keywords:**

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

- **Exceptions, try...catch statement**

  - The try statement allows you to define a block of code to be tested for errors while it is being executed.
  - The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
  - The try and catch keywords come in pairs:

```
try {
  // Block of code to try
}
catch(Exception e) {
  // Block of code to handle errors
}
```

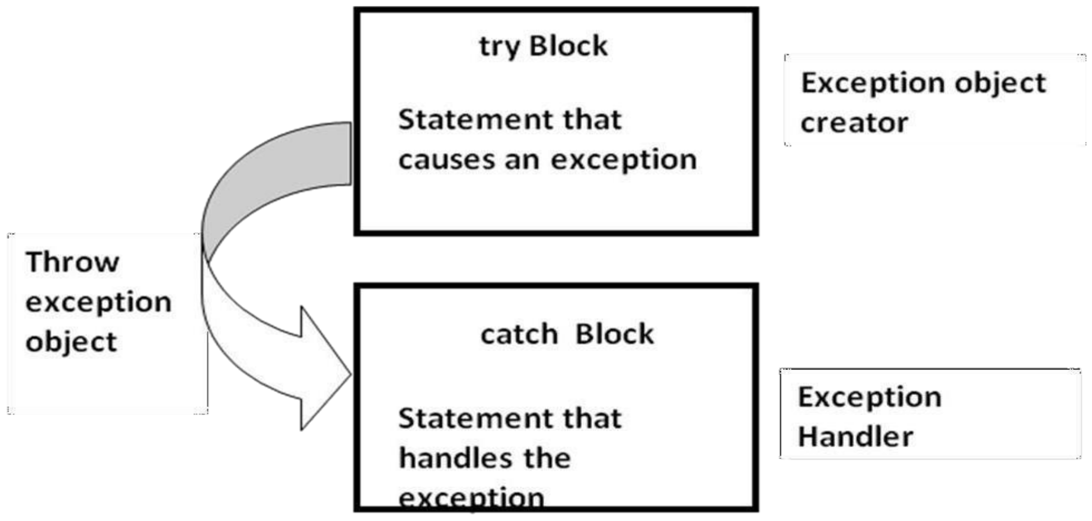


Fig. : Exception handling mechanism

Example:

```
 class TryCatchEx
{
        public  static void main(String args[])
          {
                try
                {
                        int  a=10,b=0;
                                        int x = a / b;          // Arithmetic exception
                        System.out.println("Result=" +c);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Caught an exception:" + e.getMessage());
                }
                System.out.println("End of the program");  //This line will execute
        }
}
```

Output:

```
Caught an exception: / by Zero
End of the program
```

- **Multiple catch blocks**
  - A try block can be followed by one or more catch blocks.
  - Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.
  - At a time only one exception occurs and at a time only one catch block is executed.
  - All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.
  - Syntax:

```
try
{
        //Statements
}
catch(Exception e1)
{
        …
}
catch(Exception e2)
{
        …
}
```

  - Example:

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {
          try{
           int a[]=new int[5];
           a[5]=30/0;
          }
          catch(ArithmeticException e)
            {
             System.out.println("Arithmetic Exception occurs");
            }
          catch(ArrayIndexOutOfBoundsException e)
            {
             System.out.println("ArrayIndexOutOfBounds Exception occurs");
            }
          catch(Exception e)
            {
             System.out.println("Parent Exception occurs");
            }
          System.out.println("rest of the code");
    }
}
```

Output:

```
    Arithmetic Exception occurs
    rest of the code
```

- **throw and throws keywords**

➢ throw keyword:

- We can make a program to throw an exception explicitly using throw statement.
- throw throwableInstance;
- throw keyword throws a new exception.
- An object of class that extends throwable can be thrown and caught.
- Throwable ->Exception -> MyException
- The flow of execution stops immediately after the throw statement ;
- Any subsequent statements are not executed. the nearest enclosing try block is inspected to see if it has catch statement that matches the type of exception.
- If it matches, control is transferred to that statement
- If it doesn‟t match, then the next enclosing try statement is inspected and so on. If no catch block is matched than default exception handler halts the program.
- Syntax:

```
        throw new thowable_instance;
```

- Example:

```
      throw new ArithmeticException ();
      throw new MyException();
```

Here ,new is used to construct an instance of MyException().

✓ All java‟s runtime exception s have at least two constructors:
  1) One with no parameter
  2) One that takes a string parameter.

✓ In that the argument specifies a string that describe the exception. this string is displayed when object as used as an argument to print() or println() .

✓ It can also obtained by a call to getMessage(),a method defined by Throwable class.

Example:

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }
  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.
        at Main.checkAge(Main.java:4)
        at Main.main(Main.java:12)
```

- **throws keywords**
  - The Java 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 the normal flow of the program 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 checking the code before it being used.
  - Syntax:

```
return_type method_name() throws exception_class_name{
                                   //method code
 }
```

  - **Example:**

```
import java.io.IOException;
class Testthrows1{
  void m() throws IOException{
           throw new IOException("device error");//checked exception
  }
  void n() throws IOException{
                  m();
  }
 void p(){
   try{
        n();
   }catch(Exception e){System.out.println("exception handled");}
  }

  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
 }
}
```

Output:

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

- **finally clause**
  - It can be used to handle an exception that is not caught by any of the previous catch statements.
  - It can be used to handle any exception generated within a try block.
  - It may be added immediately after try block or after the last catch block.
  - finally block is guaranteed to execute if no any exception is thrown.
  - **Use:**
    1. closing file , record set
    2. closing the connection with database
    3. releasing system resources
    4. releasing fonts
    5. Terminating network connections, printer connection etc.
    6.

We can use finally in this two way.

Syntax:1

```
try
{
        …………..
}
finally
{
        ……………...
}
```

Syntax:2

```
try
{
            ………..
}
catch(…….)
{
            …………….
}

catch (…….)
{
            ………..
}
finally
{
```

```
                    ……………
            }
```

Example:

```
class finallyEx {
   public static void main(String[] args) {
     try {
               int[] myNumbers = {1, 2, 3};
               System.out.println(myNumbers[10]);
          }
        catch  (Exception  e)  {
               System.out.println("Something went wrong.");
     }
     finally {
               System.out.println("The 'try catch' is finished.");
     }
   }
 }
```

Output:

```
Something went wrong.
The 'try catch' is finished.
```

- **Uses of exceptions**

  The ***advantages*** *of Exception Handling in Java* are as follows:

  1. Provision to Complete Program Execution
  2. Easy Identification of Program Code and Error-Handling Code
  3. Propagation of Errors
  4. Meaningful Error Reporting
  5. Identifying Error Types

- **User defined exceptions / Custom exception**
  - In java, we can create our own exceptions by extending Exception class
  - User defined exceptions are known as Custom exception.
  - It is useful when we want to create specific exception for our program.
  - You can define constructor for your Exception subclass or you can override toString() function to display customized message on catch.
  - **reasons to use custom exceptions***:*
    - To catch and provide specific treatment
    - Business logic exceptions

Example:

```
class std extends Exception
{
        private int rollno;
        std(int a)
        {
                rollno = a ;
        }
        public String toString()
        {
                return "MyException[" + rollno + "] is less than zero";
        }
}
class Student
{
        Static void sum(int a, int b) throws std
        {
          if(a<0)
         {
                throw new std(a); //throws user defined exception object
         }
         else
         {
                System.out.println(a+b);
         }
        }
        public static void main(String[] args)
        {
                try
                {
                        sum(-10,10);
                }
                catch(std me)
                {
                        System.out.println(me);
                }
        }
}
```

Output:

```
MyException["  -10 "] is less than zero
```

- **Common java Exceptions**

SR NO.	Exception Type	Cause of Exception
1	ArithmeticException	Caused by math error such as divide by zero
2	ArrayIndexOutOfBoundsException	Caused by bad array indexes
3	ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
4	FileNotFoundException	Caused by an attempt to access a nonexistent file
5	IOException	Caused by general I/O failures, such as inability to read from a file
6	NullPointerException	Caused by referencing a null object
7	NumberFormatException	Caused when a conversion between strings and number fails
8	OutOfMemoryException	Caused when there's not enough memory to allocate a new object
9	SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security

2. Concept of Multithreading, Creating thread, extending Thread class, implementing Runnable interface, life cycle of a thread, Thread priority, Thread exception handing in threads

➢ Concept of Multithreading

- **Multithreading** is a conceptual programming paradigm where a program (or process) is divided into two or more subprograms(or process),which can be implemented at the same time in parallel.
- Threads in java are subprograms of main application program and share the same memory space.
- Multithreading is similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously, to achieve a singledesire.
- We use multithreading than multiprocessing because threads use a shared memory area.
- It is mostly used in games, animation, etc.

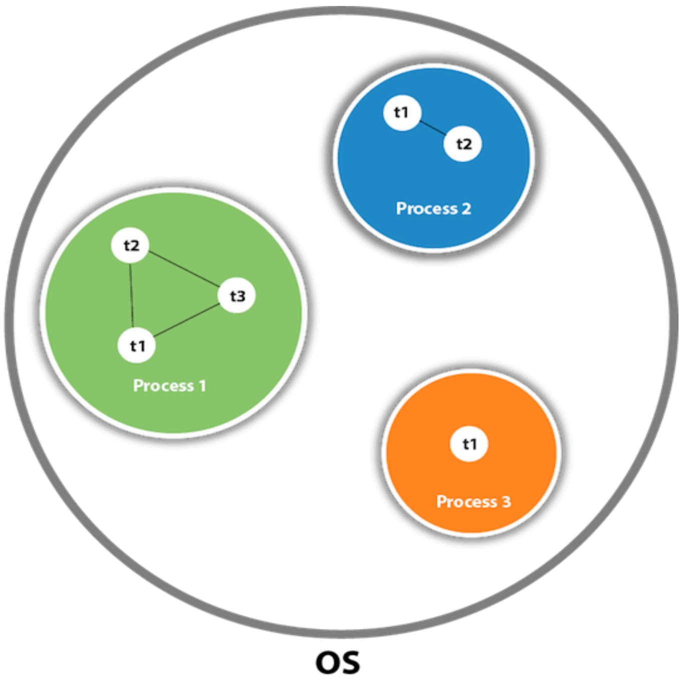
Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the 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 an exception occurs in a single thread.

Concept of thread

- Definition: “A thread is a lightweight subprocess, the smallest unit of processing.”
- It is a separate path of execution.
- Threads are independent.
- If exception occurs in one thread, it doesn't affect other threads. It uses a shared memory area.
- We can increase the speed of application using thread.
- Thread is executed inside the process as shown in figure below:



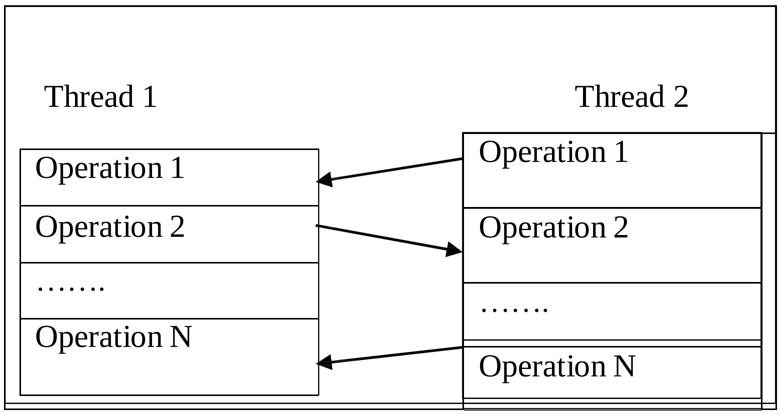
- A thread is executed along with its several operations within a single process as shown in figure below:

Operation 1
Operation 2
…….
Operation N

Single Process
Thread 1

- A multithreaded programs contain two or more threads that can run concurrently.
- Threads share the same data and code.

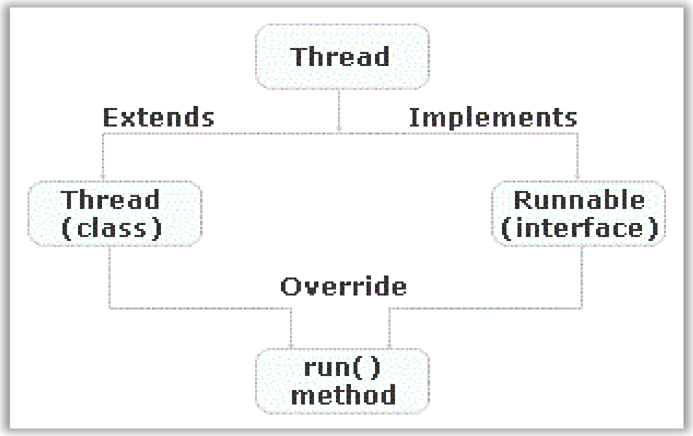
MultiThreading Concept



➢ Creating thread

Thread can be implemented through any one of two ways:

1. Extending the Java.lang.Thread class
2. Implementing Java.lang.Runnable interface



To execute a thread, it is first created and then start() method is invoked on the thread. Then thread would execute and run method would be invoked.

• <u>Extending Thread class</u>

1. Extending the java.lang.Thread class:
   a. Extend Java.lang.Thread class
   b. Override run() method in subclass from Thread class
   c. Create instance of this subclass. This subclass may call a Thread class constructor by subclass constructor.
   d. Invoke start() method on instance of class to make thread eligible for running.

Commonly used methods of Thread class:

getName()	Returns the name of the thread.
setName()	Changes the name of the thread
run()	Used to perform action for a thread
sleep()	Suspend a thread for period of time
isAlive()	Test if the thread is alive.
join()	Wait for a thread to die.
getPriority()	Returns the priority of the thread.
setPriority()	Changes the priority of the thread.
suspend()	Used to suspend the thread.
resume()	Used to resume the suspended thread.
stop()	Used to stop the thread.

Example:

```
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...
```

- **<u>Implementing Runnable interface</u>**

2. Implementing Java.lang.Runnable interface
    a. An object of this class is Runnable object.
    b. Create an object of Thread class by passing a Runnable object as argument.
    c. Invoke start() method on the instance of Thread class.

Example:

```
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); // Using the constructor Thread(Runnable r)
        t1.start();
  }
}
```

Output:

```
thread is running...
```

➢ Life cycle of a thread

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

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways.

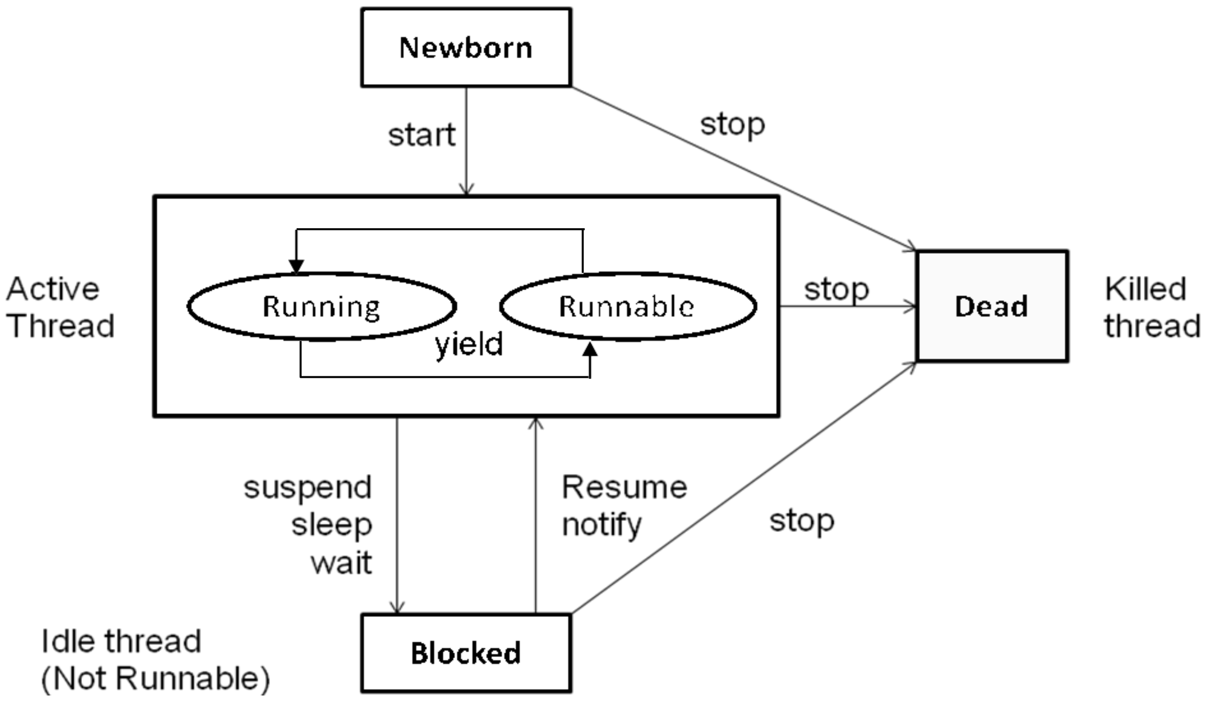


Figure: Thread Life cycle Or Thread state transition Diagram

1) New born state

- Whenever a new thread is created, it is always in the new state.
- At this time we can scheduled it for running, using start() method or kill it using stop () method.
- If we attempt to use any other method at this stage, an exception will be thrown.

2) Runnable state

- The thread is ready for execution and is waiting for the availability of the processor.
- The threads has joined waiting queue for execution.
- If all threads have equal priority, then they are given time slots for execution in roundrobin fashion. i.e. first-come, first serve manner.
- This process of assigning time to thread is known as **time-slicing.**
- If we want a thread to relinquish(leave) control to another thread of equal priority before its turn comes, then yield( ) method is used.

3) Running state

- The processor has given its time to the thread for its execution.
- The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.
- A running thread may change its state to another state in one of the following situations.
  1) When It has been suspended using suspend( ) method.
  2) It has been made to sleep( ).
  3) When it has been told to wait until some events occurs.

4) Blocked state/ Waiting

- A thread is waiting for another thread to perform a task. The thread is still alive.
- A blocked thread is considered "not runnable" but not dead and so fully qualified to run again.

5) Dead state/ Terminated

- Every thread has a life cycle. A running thread ends its life when it has completed executing its run ( ) method.
- It is natural death. However we can kill it by sending the stop message.
- A thread can be killed as soon it is born or while it is running or even when it is in "blocked" condition.

Thread class's methods:

Thread class defines several methods thathelp manage threads.

No.	**Method**	**Meaning**
1	getName( )	Obtain a thread's name.
2	setName( )	Set a thread's name.
3	getPriority( )	Obtain a thread's priority
4	setPriority( )	Set a thread's priority
5	isAlive( )	Determine if a thread is still running.
6	join( )	Wait for a thread to terminate.
7	run( )	Entry point for the thread.
8	sleep( )	Suspend a thread for a period of time.
9	start( )	Start a thread by calling run method

➢ Thread priority

- Every thread in java has its own priority.
- Thread priority are used by thread scheduler to decide when each thread should be allowed to run.
- In Java, thread priority ranges between:
    - MIN-PRIORITY(a contant of 1)
    - MAX-PRIORITY(a contant of 10)
    - NORM-PRIORITY (a contant of 5) , it is default.
- **Example:**

```
class display implements Runnable
{
        public void run()
        {
                int i= 0;
                while(i < 3)
                  System.out.println ("Hello:"+ i++);
        }
}
 public class Student
 {
        public static void main(String args[])
        {
                        display d = new display();
                        Thread t1 = new Thread(d);
                        Thread t2 = new Thread(d);
                        System.out.println("Current priority of thread t1 is:" + t1.getPriority());
                        t1.setpriority(3);
                        t1.start();
                        t2.start();
                        System.out.println("New priority of thread t1 is:" + t1.getPriority());
        }
}
```

Output:

```
F:\PG>javac Student.java
F:\PG>java Student
Current priority of thread t1 is:5
New priority of thread t1 is:3
Hello:0
Hello:1
Hello:2
Hello:0
Hello:1
Hello:2
```

➢ Thread exception handing in threads

- When we create a thread start() method performs some internal process and then calls run() method.
- The start() method does not start another thread of control but run() is not really “main” method of new thread.
- All uncaught exceptions are handled by code outside the run() method before the thread terminates.
- It is possible for a program to write a new default exception handler.
- The default exception handler is the uncaughtException method of the Thread group class.
- Whenever uncaught exception occurs in thread’s run method, we get a default exception dump which gets printed on System.err stream.

Thread class has two methods:

1. setDefaultUncaughtExceptionHandler()
2. setUncaughtExceptionHandler()

Example:

```
class MyThread extends Thread{
  public void run(){
    System.out.println("Throwing in " +"MyThread");
    throw new RuntimeException();
  }
}
public class Main {
  public static void main(String[] args) {
    MyThread t = new MyThread();
    t.start();
    try {
      Thread.sleep(1000);
    } catch (Exception x) {
      System.out.println("Caught it" + x);
    }
    System.out.println("Exiting main");
  }
}
```

Output:

```
Throwing in MyThread
Exception in thread "Thread-0" java.lang.RuntimeException
        at testapp.MyThread.run(Main.java:19)
Exiting main
```