UNIT - III

Error as human beings, we commit many errors. A software engineer may also commit several errors while designing the project or developing the code. These errors are also called 'bugs' and the process of removing them are called 'debugging'.

Errors in Java Program

There are basically three types of errors in the java program.

- Compile time errors
- Run time errors
- Logical errors

1. Compile-time Errors: These are syntactical errors found in the code, due to which a program fails to compile.

Example: forgetting s semicolon at the end of a java statement or writing a statement without proper syntax will result in compile time error. **Program:**

```
class Err
{
        public static void main(String args[])
        {
                System.out.println("Hello");
                System.out.println("Error")     // error
        }
}
```

2. Run-time Errors: These errors represent insufficiency of the computer system to execute a particular statement.

Example: Insufficient memory to store something or inability of the microprocessor to execute some statement come under run-time errors. **Program:**

```
class Err
{
        public static void main( )
        {
                System.out.println("Hello");
                System.out.println("Error");
        }
}
```

Run-time errors are not detected by the java compiler. They are detected by the JVM only at runtime.

3. Logical Errors: These errors depict flaws in the logic of the program. The programmer might be using a wrong formula or the design of the program itself is wrong. Logical errors are not detected either by Java compiler or JVM. The programmer is only responsible for them.

EXCEPTION

- An **Exception** is a run time error, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- It is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.
- When an exception occurs within a method, it creates an object.
- This object is called the exception object.
- It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred. **Major reasons why an exception Occurs**
- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

EXCEPTION HANDLING IN JAVA

- The **Exception Handling in Java** is one of the powerful *features to handle the runtime errors* so that normal flow of the application can be maintained.
- The process of converting system error messages into user friendly error message is known as 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 are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.
- If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

UNCAUGHT EXCEPTIONS

- In java, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints a exception message with the help of uncaught exception handler.
- The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.
- When an uncaught exception occurs, the JVM calls a special private method known **dispatchUncaughtException ( )**, on the Thread class in which the exception occurs and terminates the thread.

Example without Exception Handling class

```
ExceptionDemo
{
        public static void main(String[] args)
        {
                int a,b,res;
                a=20;
                b=0;
                res=a/b;
                System.out.println(res);
        }
}
```

In the above program we have not provided any exception handler, in this context the exception is caught by the default exception handler. The default exception handler displays string describing the exception.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero   at
ExceptionDemo.main(ExceptionDemo.java:8)
```

HOW TO HANDLE THE EXCEPTION

In java exception handling is managed through 5 keywords.

1: try

2: catch

3: finally

4: throw

5: throws

1: try block:

- The programmer should observe the statements in his program.
- Where there may be a possibility of exceptions. Such statements should be written inside a try block.
- A try block must be followed by catch blocks or finally block or both.

Syntax :

```
try
{
    Statements;
}
```

The try block is that even if some exception arises inside. The program will not be terminated. When jvm understands that there is an exception. It stores the exception details in an exception stack and then jumps into a catch block.

2: catch block: The programmer should write the catch block. Where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error.

Syntax :

```
catch (Exceptionclass obj)
{
    statements;
}
```

Here obj is automatically to refer to the exception stack where the details of the exception are available.

Methods to print the Exception information:

We can display the exception details using any one of the following ways.

1. printStackTrace ()– This method prints exception information in the format of Name of the exception: description of the exception, stack trace.

//program to print the exception information using printStackTrace() method

```
import java.io.*; class Sample
{
        public static void main (String[] args)
        {
                int a=20, b=0;
        try
                {
                        System.out.println(a/b);
                }
catch(ArithmeticException e)
                {
                        e.printStackTrace();
                }
        }
}
```

OUTPUT : java.lang.ArithmeticException: / by zero at Sample1.main(Sample1.java:17)

2. toString () – This method prints exception information in the format of Name of the exception: description of the exception.

//program to print the exception information using toString() method import java.io.*;

```
class Sample
{
        public static void main (String[] args)
        {
                int a=20, b=0;
        try
                {
                        System.out.println(a/b);
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e.toString());
                }
        }
}
```

OUTPUT : java.lang.ArithmeticException: / by zero

3.getMessage() -This method prints only the description of the exception.

//program to print the exception information using getMessage() method

```
import java.io.*;
class Sample
 {
        public static void main (String[] args)
        {
                int a=20, b=0;
                try
                {
                        System.out.println(a/b);
                        System.out.println("This is Try Block");
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e.getMessage());
                        System.out.println("This is getMessage method");
                }
                System.out.println("Close Try-Catch Block");
        }
}
```

OUTPUT:

```
/ by zero
This is getMessage method
Close Try-Catch Block
```

Multiple catch Statements: Multiple catch blocks handle the situation where more than one exception could be raised by a single piece of code. In this case we can specify two or more catch block statements. Multible catch statements are arranged in respective order of exceptions that araised by the try block.

Syntax :

```
try
{
    statements;
}
catch(Exceptionclass1 obj)
{
    statements;
}
 catch(Exceptionclass2 obj)
{
    statements;
}
   .
   .
catch(ExceptionclassN obj)
{
    statements;
}
```

Program :

```
class Ex2
{
    public static void main(String[] args)
    {
            try
          {
                    System.out.println("Open file”);
                    int n=args.length;
                    System.out.println("a value is:"+a);
                    int a=45/n;
                System.out.println("a value is:"+a);
int b1[]={10,20,30};
System.out.println(b1[0]);
                    System.out.println(b1[1]);
                    System.out.println(b1[2]);
                    System.out.println(b1[50]);
                    System.out.println("Hello World!");
            }
            catch(ArithmeticException e)
            {
                    e.printStackTrace();
            }
            catch(ArrayIndexOutOfBoundsException aio)
            {
                    aio.printStackTrace();
            }
           System.out.println("Hello World!");
     }
}
```

Note : Even though there is possibility for several exceptions in try block, at a time only one exception will be raised.

OUTPUT :

```
javac Ex2.java
Java Ex2
Open file
a value is: 0
java.lang.ArithmeticException: / by zero at Ex2.main(Ex2.java:10) Hello
World
```

NESTED TRY STATEMENTS

The try block within a try block is known as nested try block in java.

Why 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  {
        statement 1;
      statement 2;
        try
        {
              statement 1;
          statement 2;
        }
        catch(Exception e)
        {
             ……….
        }
    }
    catch(Exception e)
    {
         ……..
    }
```

Example

```
class NestedTry
{
    public static void main(String args[])
    {
        try
            {
            try
              {
                  int a[]=new int[5];
            a[5]=4;
              }
              catch(ArrayIndexOutOfBoundsException e)
              {
                    System.out.println(e);
              }
         }
        catch(Exception e)
        {
              System.out.println("handeled");
         }
```

```
        System.out.println("normal flow..");
    }
}
```

3: finally:

- 'Finally' is a keyword of java used to create a block of code that will be executed after a try / catch block whether exception is handled or not.
- If an exception occurs then the catch block executes after that finally block will be executed.
- If no exception occurs, then finally block executed.
- In case there is an exception, but there is no matching catch block. Then the code execution skips the rest of code and executes the finally block.
- A finally block appears at the end of catch block.
- The finally block follows a try block or a catch block.

Syntax:

```
try
{
    statements;
}
catch(Exceptionclass obj)
{
   statements;
}
finally
{
    statements;
}
```

Program:

```
class ExceptionDemo
{ public static void main(String[] args)
        {
            try
            {
                int a=20,b=0,res; res=a/b;
                System.out.println(res);
            }
            catch(ArithmaticException e )
            {
                System.out.println("Division by Zero");
            } finally
            {
                System.out.println("Hellow This is Finally Block");
            }
```

```
                System.out.println("Program Over");
        }
}
```

4.THROW KEYWORD:

- The throw keyword is used to explicitly throw a single exception.
- When an exception is thrown, the flow of program execution transfers from the try block to the catch block. We use the throw keyword within a method.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.
- The flow of exception stops immediately after the throw statement and any subsequence statements are not executed.

Syntax: throw new ExceptionType("content"); **Example:**

```
import java.lang.*; class
Sample
{
        public static void main(String args[])
      {
try
                {
                        throw new ArithmeticException (" This is my own Exception");
System.out.println(" This is try Block");
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e.getMessage());
                }
                System.out.println(" Program Over ");
        }
}
```

OUTPUT : javac Sample.java

Java Sample

This is my own Exception

Program Over

5.THROWS KEYWORD

We use the throws keyword in the method declaration to declare the type of exceptions that might occur within it.

Syntax:

```
accessModifier returnType methodName() throws ExceptionType1, ExceptionType2 …
{
        // code
}
```

Example:

```
import java.lang.*; class
Throws1
{
 static void display()throws ArithmeticException
 {
        int x=10;
int y=0;        int
res=x/y;
        System.out.println(res);
 }
 public static void main(String args[])
 {
        try
        {
                display();
        }
        catch(ArithmeticException e)
        {
                System.out.println(e.getMessage());
        }
 }
}
```

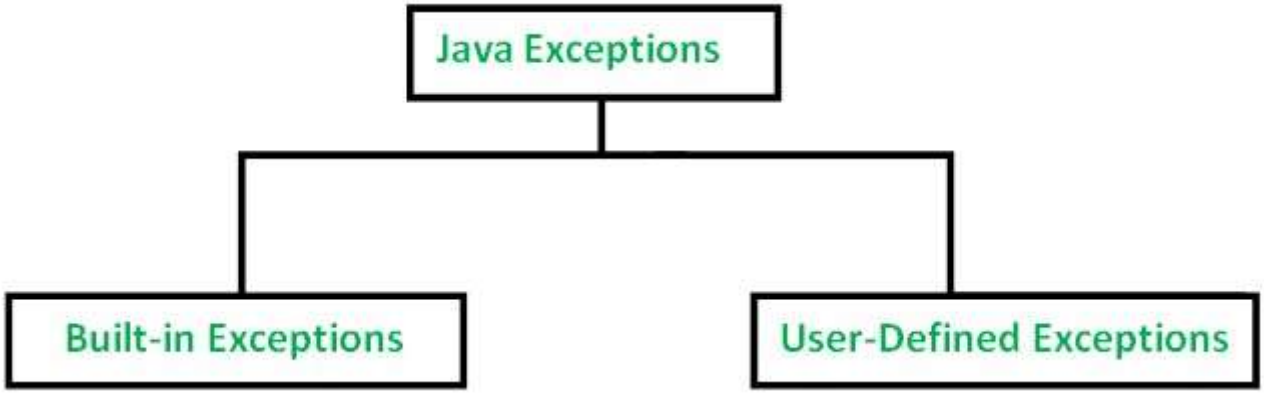
OUTPUT: javac Throws1.java

Java Throws1

/ by Zero

EXCEPTION TYPES

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Built-in Exceptions

it is a pre-defined exception in the Exception class of java.lang package. These can be further divided into 2 types.

1. Checked Exception
2. Unchecked Exception

Checked Exception: Java developers have identified certain exceptions which must be made known to the programmer at the compile time itself. Such exceptions are listed and the compiler checks whether any of the listed exceptions occur in a program during compilation. Such exceptions are called checked exception. checked exceptions defined in java.lang package.

Example:

- ClassNotFoundException
- File Not Found Exception
- IOException
- Instantiation Exception
- InterruptedException
- SQLException

1.ClassNotFoundException: This Exception is raised when we try to access a class whose definition is not found class ClassNotFoundException_Demo

```
{ public static void main(String[] args)
        {
                try
                {
                        Class.forName("Class1");   // Class1 is not defined
                }
                catch(ClassNotFoundException e)
                {
                        System.out.println(e);
                        System.out.println("Class Not Found...");
                }
        }
}
```

2. FileNotFoundException: This Exception is raised when a file is not accessible or does not open.

```
import java.io.File;
import java.io.FileNotFoundException; import
java.io.FileReader;
class File_notFound_Demo
{ public static void main(String args[])
        {
                try
                {
                        File file = new File("E://file.txt");   // Following file does not exist
        FileReader fr = new FileReader(file);
                }
                catch (FileNotFoundException e)
                {
                        System.out.println("File does not exist");
                }
         }
}
```

3.IOException : When any of the input or output gets terminated abnormally and operation gets failed then it causes IO Exception.

4.Instantiation Exception : An object is called an instance of a class. Now, a class can not be instantiated if it is an abstract class, an interface, or a void class. InstantiationException is thrown when we try to create an instance of a class using the newInstance method, but that class cannot be instantiated. A simple

scenario where a class cannot be instantiated is when the class which we want to instantiate is an abstract class.

5. Interrupted Exception: Threads are used in Java to improve the efficiency of the code by allowing it to do multiple things together at the same time. Threads are extended from the predefined Thread class in Java. It provides various functions like sleep which allows us to temporarily stop the execution of the thread (for the specified number of milliseconds). InterruptedException is thrown when a thread is sleeping, waiting, or occupied and it is disturbed.

```
class interruptException extends Thread
{
   public void run()
   {
try
      {
         Thread.sleep(1000);
      }
      catch (InterruptedException e)
      {
        System.err.println(e);
      }
   }
   public static void main()
   {
      interruptException obj = new interruptException();
obj.start();        obj.interrupt();
   }
}
```

6. SQL Exception : *SQLException* is thrown if there is an error in database access or other database errors. To access the Database, we use various functions like Connection, DriverManager, and getConnection.

```
import java.sql.Connection; import
java.sql.DriverManager; import
java.sql.SQLException; public class
sqlexception
{
   public static void main() throws SQLException
   {
      Connection conn = DriverManager.getConnection("Database_URL");
   }
}
```

2. Unchecked Exception: Exceptions that are not listed in the list are identified during runtime. Such exceptions are called unchecked exceptions. Unchecked exceptions defined in java.lang package.

Note: In real time application mostly we can handle un-checked exception.

Example:

1. ArithmeticException
2. ArrayIndexOutofBoundsException
3. NullPointerException
4. StringIndexOutOfBoundsException
5. NumberFormateException
6. ClassCastException
7. ArrayStoreException
8. IllegalThreadStateException

1.ArithmeticException: It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
class ArithmeticException_Demo
{
        public static void main(String args[])
    {
try
                {
                        int a = 30, b = 0;
                         int c = a/b;  // cannot divide by zero
                        System.out.println ("Result = " + c);
                }
                catch(ArithmeticException e)
                {
                        System.out.println ("Can't divide a number by 0");
                }
        }
}
```

2.ArrayIndexOutOfBoundsException: It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

```
class ArrayIndexOutOfBoundsException_Demo
{
        public static void main(String args[])
    {
try
                {
                        int a[] = {10,20,30};
                        System.out.println(a[10]);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println ("Array Index is Out Of Bounds");
                }
        }
}
```

3.NullPointerException: This exception is raised when referring to the members of a null object. Null represents nothing.

```
class NullPointer_Demo
{
        public static void main(String args[])
    {
try
                {
                        String a = null; //null value
                        System.out.println(a.charAt(0));
                }
                catch(NullPointerException e)
                {
                        System.out.println("NullPointerException..");
                }
        }
}
```

4.StringIndexOutOfBoundsException: It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string.

```
class StringIndexOutOfBound_Demo
{
        public static void main(String args[])
      {
try
                {
                        String a = "This is like chipping "; // length is 22
              char c = a.charAt(24); // accessing 25th element
                        System.out.println(c);
                }
                catch(StringIndexOutOfBoundsException e)
                {
                        System.out.println("StringIndexOutOfBoundsException");
                }
        }
}
```

5. NumberFormatException: This exception is raised when a method could not convert a string into a numeric format.

```
class NumberFormat_Demo
{
        public static void main(String args[])
      {
try
                {
                        int num = Integer.parseInt ("akki") ; // "akki" is not a number
                        System.out.println(num);
                    }
                catch(NumberFormatException e)
                {
                        System.out.println("Number format exception");
                }
        }
}
```

6. ClassCastException: Class Cast Exception Type casting is changing the type from one type to another. Casting a class is changing its type. *ClassCastException* is thrown by JVM when we try to cast a class from one type to another, and it violates some rules.

7. ArrayStoreException: Array Store Exception *ArrayStoreException* is thrown by JVM when we try to store the wrong type of object in the array of objects.

8. IllegalThreadStateException: IllegalStateException is the child class of RuntimeException and hence it is an unchecked exception. This exception is rise explicitly by programmer or by the API developer to indicate that a method has been invoked at the wrong time. Generally, this method is used to indicate a method is called at an illegal or inappropriate time.

Example: After starting a thread we are not allowed to restart the same thread once again otherwise we will get Runtime Exception saying IllegalStateException.

USER DEFINED EXCEPTIONS OR CUSTOMIZED EXCEPTION

Sometimes we need to create our own exception to meet the program requirements, these exceptions are called user defined exception or customized exception **Example:**

InsufficientAmountException

DailyLimitCrossedException

InvalidAgeException

Rules to design user defined Exception

- Create our own class.
- Make our class as subclass of Exception.
- Declare parameterized constructor with string variable.
- call super class constructor by passing string variable in parameterized constructor.

Example : class InsufficientAmountException
extends Exception

```
{
        InsufficientAmountException(String s)
        {
                super(s);
        }
}
```

```
class Account
{
   static void withdraw(int amount) throws InsufficientAmountException
  {
     If(amount>20000)
          throw new InsufficientAmountException("You do not have sufficient amount to withdraw");
else
          System.out.println(" please withdraw money ");


   }
  public static void main(String args[])
  {
              try
              {
                        withdraw(30000);
              }
              catch(InsufficientAmountException e)
              {
                        System.out.println(e);
              }
              System.out.println(" Thanks for visiting ");
    }
}
```

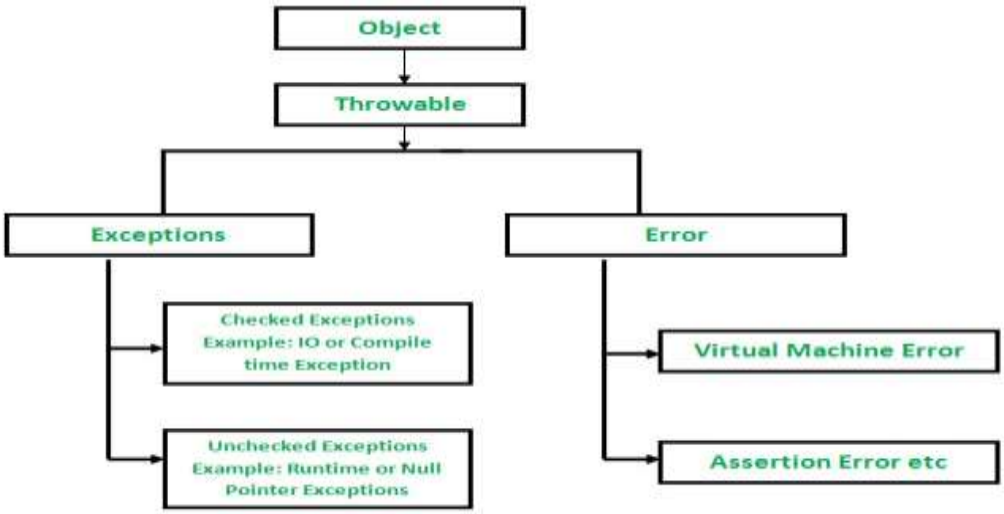
OUTPUT : javac Account.java

java Account

InsufficientAmountException : You do not have sufficient amount to withdraw **EXCEPTION**

HIERARCHY IN JAVA

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below



Difference between throw and throws

throw	throws
1. Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	1. Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.

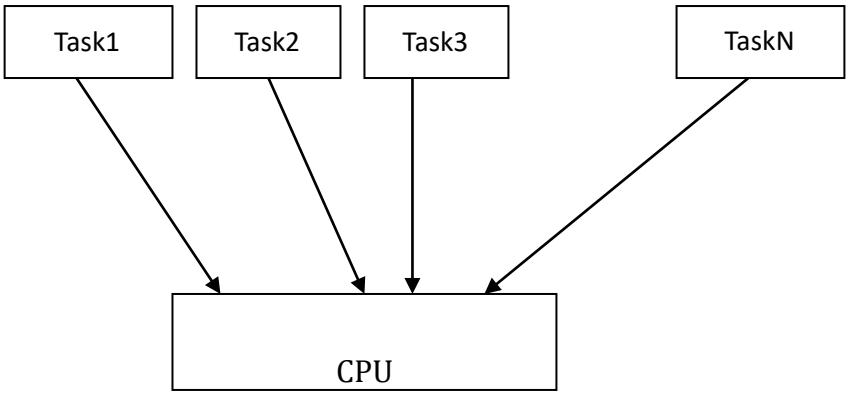
2. The throw keyword is followed by an instance of Exception to be thrown.
Example:
throw new ArithmeticException("Hello");
	2. The throws keyword is followed by class names of Exceptions to be thrown.
	Example:
	void display()throws ArithmeticException
	{
	}

3. throw is used within the method.	3. throws is used with the method signature.
4. We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	4. We can declare multiple exceptions using throws keyword that can be thrown by the method. **example**, main() throws IOException, SQLException.
5. One can only propagate the unchecked exceptions using the throw keyword. It means that no checked exception can be propagated when we use the throw keyword.	5. When we use the throws keyword, we can declare both unchecked and checked exceptions. The checked expression must always use the throws keyword for propagation followed by a specific name of the exception class.

MULTITHREADING

Before starting the thread concept, let us begin with a known multitasking process.

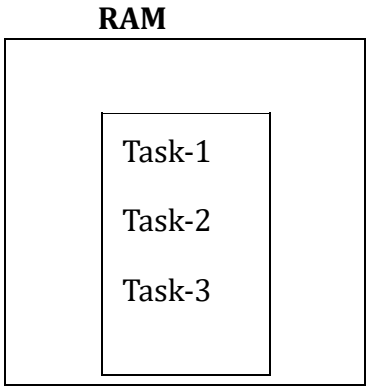
Multitasking: When a single ***processor (CPU)*** does many jobs (program, threads, process, task) simultaneously, it is called ***multitasking***. The CPU switches between these activities often so the user can engage with each program simultaneously to execute multitasking. Each task has its own set of variables and separate memory location for them.



Multitasking has Two Types:

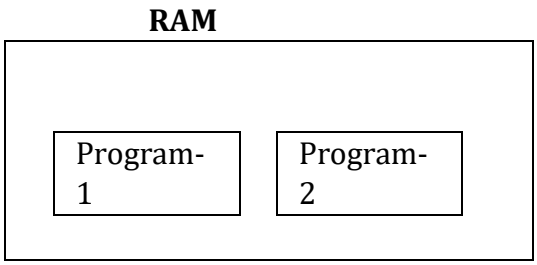
1. Process-based Multitasking
2. Thread-based Multitasking

1. Process-based Multitasking:



All the functionality which are supposed to be executed simultaneously are in same domain in RAM is called Process-Based Multitasking.

2 : Thread-Based Multitasking :



Here two programs are in different domain in ram is called Thread-Based Multitasking.

Java supports only Thread-Based Multitasking but not Process-Based Multitasking.

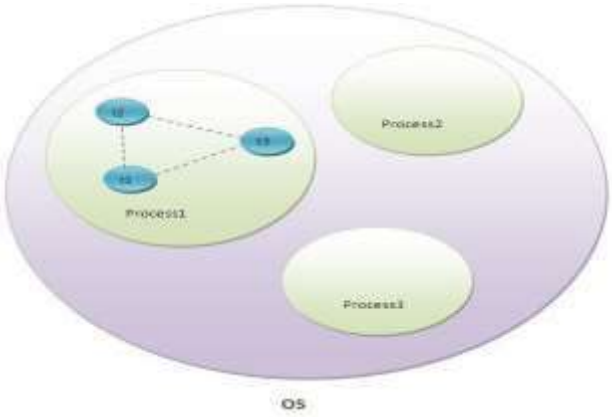
Difference between Process-Based Multitasking and Thread-Based Multitasking

Process-Based Multitasking	Thread-Based Multitasking
In process-based multitasking, two or more processes and programs can be run concurrently.	In thread-based multitasking, two or more threads can be run concurrently.
In process-based multitasking, a process or a program is the smallest unit.	In thread-based multitasking, a thread is the smallest unit.
The program is a bigger unit.	Thread is a smaller unit.
Process-based multitasking requires more overhead.	Thread-based multitasking requires less overhead.
The process requires its own address space.	Threads share the same address space.
The process to Process communication is expensive.	Thread to Thread communication is not expensive.
Here, it is unable to gain access over the idle time of the CPU.	It allows taking gain access over idle time taken by the CPU.
It is a comparatively heavyweight.	It is comparatively lightweight.
It has a faster data rate for multi-tasking because two or more processes/programs can be run simultaneously.	It has a comparatively slower data rate multitasking.

Thread Concept in Java

A Thread is a:

- Feature through which we can perform multiple activities within a single process.
- Lightweight process.
- Series of executed statements.
- Nested sequence of method calls.



As shown in the above figure, 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.

Note: At a time only one thread is executed.

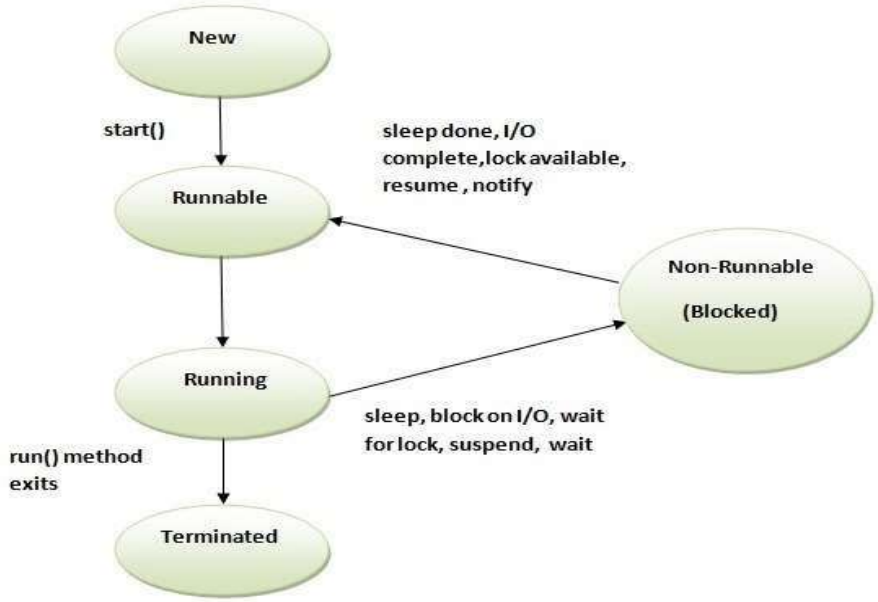
Difference between Multitasking and Multithreading.

Multitasking	Multithreading
When a single processor does many jobs (program, threads, process, task) at the same time, it is referred to as multitasking.	Multitasking occurs when the CPU does many tasks, such as a program, process, task, or thread.
A CPU may perform multiple tasks at once by using the multitasking method.	Multithreading allows a CPU to generate numerous threads from a job and process them all at the same time.
The system must assign different resources and memory to separate programs that are running concurrently in multitasking.	The system assigns a single memory block to each process.
CPU switches between programs frequently.	CPU switches between the threads frequently.
It is comparatively slower in execution.	It is comparatively faster in execution.
A user may easily run several jobs off of their CPU at once.	A CPU has the ability to split a single program into several threads to improve its functionality and efficiency.
The process of terminating a task takes comparatively more time.	It requires considerably less time to end a process.

Java Thread Model (Life Cycle of a Thread)

A thread can be in one of the five states in the thread. The life cycle of the thread is controlled by JVM.
The thread states are as follows:

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



New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

Running

The thread is in running state if the thread scheduler has selected it.

Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

Terminated

A thread is in terminated or dead state when its run() method exits.

CREATING THREADS

The java programming language provides two methods to create threads.

1. Extending Thread class
2. Implementing Runnable interface **Thread class:**

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface **Commonly used Constructors of Thread class:**

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void suspend():** is used to suspend the thread(depricated).
15. **public void resume():** is used to resume the suspended thread(depricated).
16. **public void stop():** is used to stop the thread(depricated).

Extending Thread class

To create a thread using Thread class, follow the step given below.

1. Thread class is defined in the package java.lang.*;
2. Create a class that extends Thread class.

   Example : class Sample extends Thread
   {
   }
3. Write a run( ) method. Override the run( ) method with the code that is to be executed by the thread.

   The run( ) method must be public while overriding.

   Example : public void run()
   {
   Statements;
   }
4. Create the object of the newly created class in the main( ) method **Example :** Sample s = new Sample();
5. Call the start( ) method on the object created in the above step.

   Example : s.start(); class

SampleThread extends Thread

```
{
        public void run()
        {
            System.out.println("Thread is under Running...");
      for(int i= 1; i<=10; i++)
                {
                    System.out.println("i = " + i);
                }
        }
}
public class My_Thread_Test
{
        public static void main(String[] args)
        {
                SampleThread st = new SampleThread();
                System.out.println("Thread about to start...");
                st.start();
        }
}
```

Implementing Runnable interface

To create a thread using Thread class, follow the step given below.

1. Thread class is defined in the package java.lang.*;
2. Create a class that implements Runnable interface.

   Example : class Sample extends Thread
   {
   }
3. Write a run( ) method. Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

   Example : public void run()
   {
   Statements;
   }
4. Create the object of the newly created class in the main( ) method **Example :** Sample s = new Sample();
5. Create the Thread class object by passing above created object as parameter to the Thread class constructor.

   Example : Thread t= new Thread(s);
6. Call the start( ) method on the object created in the above step.

   Example : s.start(); class

SampleThread implements Runnable

```
{
        public void run()
        {
                System.out.println("Thread is under Running...");
                for(int i= 1; i<=10; i++)
                {
                        System.out.println("i = " + i);
                }
        }
}
public class My_Thread_Test
{
        public static void main(String[] args)
        {
                SampleThread st = new SampleThread();
                Thread t = new Thread(st)
                System.out.println("Thread about to start...");
                t.start();
```

```
        }
}
```

Thread Priorities

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

Three constants defiend in Thread class:

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

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

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

Output:

```
    running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

Synchronizing Threads

Synchronization

Synchronization is the capability of control the access of multiple threads to any shared resource. Synchronization is better in case we want only one thread can access the shared resource at a time.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronizations.

1. Mutual Exclusive 2.

Synchronized method.

3. Synchronized block.
4. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by two ways in java:

1. by synchronized method
2. by synchronized block

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent.

Example:

```
class Table
{
        void printTable(int n)
        {
                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);
        }
}
class TestSynchronization
{
        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
100
10
200
15
300
20
400
25
500
```

Java synchronized method

- 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.

Example:

```
class Table
{
        synchronized void printTable(int n)
        {
                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);
        }
}
class TestSynchronization
{
        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

- 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.
- Points to remember for Synchronized block
- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
synchronized (object reference expression)
{
    //code block
}
```

Example of synchronized block

```
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 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

Interthread Communication

- Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**.
- In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true.
- That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task.
- The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

Method	**Description**
void wait( )	It makes the current thread to pause its execution until other thread in the same monitor calls notify( )
void notify( )	It wakes up the thread that called wait( ) on the same object.
void notifyAll()	It wakes up all the threads that called wait( ) on the same object.

Let's look at an example problem of producer and consumer.

- The producer produces the item and the consumer consumes the same.
- But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced.
- So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same.
- Here we use the inter-thread communication to implement the producer and consumer problem.

Example class

```
ItemQueue
{
 int item;
 boolean valueSet = false;  synchronized
int getItem()
 {
            while (!valueSet)
            try
           {
                  wait();
           }
           catch (InterruptedException e)
          {
                  System.out.println("InterruptedException caught");
          }
          System.out.println("Consummed:" + item);
          valueSet = false;
             try
           {
                   Thread.sleep(1000);
             }
           catch (InterruptedException e)
          {
                   System.out.println("InterruptedException caught");
          }
          notify();
          return item;
 }
synchronized void putItem(int item)
{
          while (valueSet)
          try
           {
                  wait();
             }
           catch (InterruptedException e)
           {
                  System.out.println("InterruptedException caught");
```

```
            }
        this.item = item;
        valueSet = true;
            System.out.println("Produced: " + item);
            try
         {
                 Thread.sleep(1000);
            }
         catch (InterruptedException e)
         {
                System.out.println("InterruptedException caught");
            }
            notify();
 }
}
class Producer implements Runnable
{
            ItemQueue itemQueue;
            Producer(ItemQueue itemQueue)
            {
                this.itemQueue = itemQueue;
        new Thread(this, "Producer").start();
            }
            public void run()
            {
                int i = 0;
        while(true)
                 {
                        itemQueue.putItem(i++);
                   }
            }
}
class Consumer implements Runnable
{
            ItemQueue itemQueue;
            Consumer(ItemQueue itemQueue)
            {
                this.itemQueue = itemQueue;
        new Thread(this, "Consumer").start();
            }
            public void run()
```

```
        {
                while(true)
                {
                        itemQueue.getItem();
                }
        }
}
class ProducerConsumer
{
        public static void main(String args[])
        {
            ItemQueue itemQueue = new ItemQueue();
            new Producer(itemQueue);
        new Consumer(itemQueue);
        }
}
```

Output:

```
Produced: 0
Consummed:0
Produced: 1
Consummed:1
Produced: 2
Consummed:2
Produced: 3
Consummed:3
Produced: 4
Consummed:4
Produced: 5
Consummed:5
Produced: 6
Consummed:6
Produced: 7
Consummed:7
Produced: 8
Consummed:8
```