

## UNIT - 3

### EXCEPTION HANDLING

#### What is Exception

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. This leads to the abnormal termination of the program, which is not always recommended.

#### 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 Handling Fundamentals :

An exception may occur due to the following reasons. They are.

- Invalid data as input.
- Network connection may be disturbed in the middle of communications
- JVM may run out of memory.
- File cannot be found/opened.

These exceptions are caused by user error, programmer error, and physical resources. Based on these, the exceptions can be classified into three categories.

- **Checked exceptions** – A checked exception is an exception that occurs at the compile time, also called as compile time (static time) exceptions. These exceptions cannot be ignored at the time of compilation. So, the programmer should handle these exceptions.
- **Unchecked exceptions** – An unchecked exception is an exception that occurs at run time, also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- **Errors** – Errors are not exceptions, but problems may arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.
- **Error:** An Error indicates serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

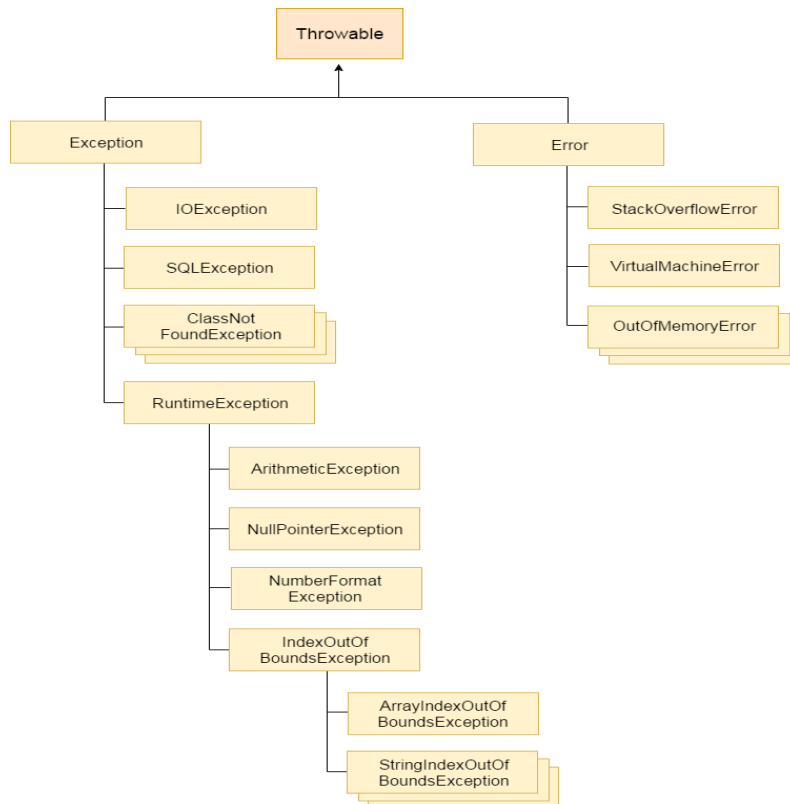
#### EXCEPTION HIERARCHY

The java.lang.Exception class is the base class for all exception classes. All exception and errors types are sub classes of class Throwable, which is base class of hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. Null Pointer Exception is an example of such an exception. Another branch, Error are used by the Java run-time system(JVM) to indicate errors having to do with the runtime environment itself(JRE). Stack Overflow Error is an example of such an error.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment.

Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: **IOException class** and **RuntimeException Class**.



### Exceptions Methods

Method	Description
public String getMessage()	Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
public Throwable getCause()	Returns the cause of the exception as represented by a Throwable object.
public String toString()	Returns the name of the class concatenated with the result of getMessage().
public void printStackTrace()	Prints the result of toString() along with the stack trace to System.err, the error output stream.
public StackTraceElement [] getStackTrace()	Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
public Throwable fillInStackTrace()	Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

### Advantage of Exception Handling:

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.

### Exception handling in java uses the following Keywords

1. try
2. catch
3. finally

4. throw
5. throws

## try Block

---

Enclose the code that might throw an exception within a *try* block. If an exception occurs within the *try* block, that exception is handled by an exception handler associated with it. The *try* block contains at least one *catch* block or *finally* block.

### The syntax of java try-catch

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

### The syntax of a try-finally block

```
try{  
  
//code that may throw exception  
  
}finally{ }
```

### Nested try block

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

```
public class NestedTryBlock {  
  
    public static void main(String args[]) {  
        try {  
            try {  
                System.out.println(" This gives divide by zero error");  
                int b = 39 / 0;  
            } catch (ArithmeticException e) {  
                System.out.println(e);  
            }  
            try {  
                System.out.println(" This gives Array index out of bound exception");  
                int a[] = new int[5];  
            }  
        }  
    }  
}
```

```

        a[5] = 4;

    } catch (ArrayIndexOutOfBoundsException e) {

        System.out.println(e);

    }

    System.out.println("other statement");

} catch (Exception e) {

    System.out.println("handeled");

}

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

}

}

```

### catch Block

---

Java *catch* block is used to handle the Exception. It must be used after the *try* block only. You can use multiple *catch* block with a single try.

Syntax:

```

try

{

    //code that cause exception;

}

catch(Exception_type e)

{

    //exception handling code

}

```

### catch Block Examples

Let's demonstrate the usage of catch block using *ArithmeticException* type.

**Example 1:** *ArithmeticException* exception type example

```

public class Arithmetic {

    public static void main(String[] args) {

        try {

            int result = 30 / 0; // Trying to divide by zero

```

```
} catch (ArithmeticException e) {  
  
    System.out.println("ArithmeticException caught!");  
  
}  
  
System.out.println("rest of the code executes");  
  
}  
  
}
```

Output:

ArithmeticException caught!

rest of the code executes

### throw Keyword

It is possible for your program to throw an exception explicitly, using the *throw* statement. The general form of throw is shown here:

```
throw ThrowableInstance;
```

### throws Keyword

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 normal flow can be maintained.

### The syntax of java throws

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

### finally Block

- Java *finally* block is a block that is used to execute important code such as closing connection, stream etc.
- Java *finally* block is always executed whether an exception is handled or not.
- Java *finally* block follows try or catch block.
- For each try block, there can be zero or more catch blocks, but only one *finally* block.
- The *finally* block will not be executed if program exits(either by calling *System.exit()* or by causing a fatal error that causes the process to abort).

## Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

### Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An 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 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.

### Termination Model :

In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.

## **Resumptive Model :**

The alternative of termination model is resumptive model. In resumptive model, the exception handler is expected to do something to stable the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.

In resumptive model we may use a method call that want resumption like behavior. We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.

## **Uncaught Exceptions in Java :**

In java, assume that, 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.

Java programming language has a very strong exception handling mechanism. It allow us to handle the exception use the keywords like try, catch, finally, throw, and throws.

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:**

```
import java.util.Scanner;
public class UncaughtExceptionExample {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
        System.out.println("Enter the a and b values: ");
        int a = read.nextInt();
        int b = read.nextInt();
        int c = a / b;
        System.out.println(a + "/" + b + " = " + c);
    }
}
```

## **Java Nested try block :**

In Java, using a try block inside another try block is permitted. It is called as nested try block.

Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the inner try block can be used to handle `ArrayIndexOutOfBoundsException` while the outer try block can handle the `ArithmeticException` (division by zero).

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:**

```
//main try block
try
{
    statement 1;
    statement 2;
    //try catch block within another try block
    try
```

```

{ statement 3;
statement 4;
//try catch block within nested try block
try
{
statement 5;
statement 6;
}
catch(Exception e2)
{
//exception message
}}
catch(Exception e1)
{
//exception message
}}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}

```

### **Java Nested try Example**

Let's see an example where we place a try block within another try block for two different exceptions.

```

public class NestedTryBlock
{
public static void main(String args[])
{
//outer try block
try{
//inner try block 1
try{
System.out.println("going to divide by 0");
int b =39/0;
}
//catch block of inner try block 1
catch(ArithmeticException e)
{
System.out.println(e);
}
//inner try block 2
try{
int a[]=new int[5];
//assigning the value out of array bounds
a[5]=4;
}
//catch block of inner try block 2

```



```

catch(ArrayIndexOutOfBoundsException e)
{
System.out.println(e);
}
System.out.println("other statement");
}
//catch block of outer try block
catch(Exception e)
{
System.out.println("handled the exception (outer catch)");
}
System.out.println("normal flow..");
}}

```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

### **Throw keyword**

The Java throw keyword is used to explicitly throw an exception. The general form of throw is shown below:

#### **throw ThrowableInstance;**

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non Throwable classes, such as String and Object, cannot be used as exceptions.

### **There are two ways to obtain a Throwable object:**

1. using a parameter in a catch clause
2. creating one with the new operator.

### **The following program explains the use of throw keyword.**

```

public class TestThrow1
{
    static void validate(int age){
    try{
        if(age<18)
            throw new ArithmeticException("not valid");
        else
            System.out.println("welcome to vote");
        }
        Catch(ArithmeticException e)
        {
            System.out.println("Caught inside ArithmeticExceptions.");throw e; //
            rethrow the exception
        }
    }
    public static void main(String args[]){
    try{
        validate(13);
    }
    Catch(ArithmeticException e)
    {

```

```

        System.out.println("ReCaught ArithmeticExceptions.");
    }
}

```

The flow of execution stops immediately after the throw statement and any subsequent statements that are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

### **The Throws/Throw Keywords**

If a method does not handle a checked exception, the method must be declared using the throws keyword. The throws keyword appears at the end of a method's signature.

The difference between throws and throw keywords is that, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

### **The following method declares that it throws a Remote Exception –**

#### **Example**

```

import java.io.*;
public class throw_Example1 {
    public void function(int a) throws RemoteException {
        // Method implementation throw
        new RemoteException();
    }
} // Remainder of class definition

```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an ArithmeticException –

```

import java.io.*;
public class throw_Example2 {
    public void function(int a) throws RemoteException, ArithmeticException {
        // Method implementation
    }
} // Remainder of class definition

```

### **The Finally Block**

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of the occurrence of an Exception. A finally block appears at the end of the catch blocks that follows the below syntax.

```

Syntax
try {
    // Protected code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
    // Catch block
}
finally {
    // The finally block always executes.
}

```

#### **Example**

```

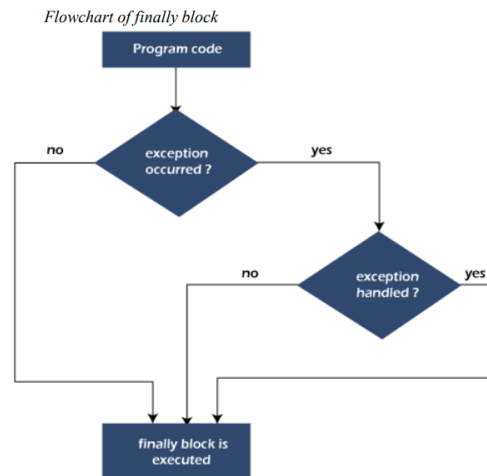
public class Finally_Example {
    public static void main(String args[]) {
        try {

```

```

int a,b;
    a=0;
    b=10/a;
}    catch    (ArithmeticException    e)    {
System.out.println("Exception thrown :" + e);
}finally {
System.out.println("The finally block is executed");
}
}
}

```



## BUILT-IN EXCEPTIONS:

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

Exceptions	Description
Arithmetic Exception	It is thrown when an exceptional condition has occurred in an arithmetic operation.
Array Index Out Of Bound Exception	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.
ClassNotFoundException	This Exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	This Exception is raised when a file is not accessible or does not open.
IOException	It is thrown when an input-output operation failed or interrupted.
InterruptedException	It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
NoSuchFieldException	It is thrown when a class does not contain the field (or variable) specified.
NoSuchMethodException	It is thrown when accessing a method which is not found.

NullPointerException	This exception is raised when referring to the members of a null object. Null represents nothing.
NumberFormatException	This exception is raised when a method could not convert a string into a numeric format.
RuntimeException	This represents any exception which occurs during runtime.
StringIndexOutOfBoundsException	It is thrown by String class methods to indicate that an index is either negative than the size of the string

**Arithmetic Exception :** 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");
}}}
```

**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 ArrayIndexOutOfBounds_Demo {
public static void main(String args[])
{
try {
int a[] = new int[5];
a[6] = 9; // accessing 7th element in an array of size 5
}
catch (ArrayIndexOutOfBoundsException e) {
System.out.println("Array Index is Out Of Bounds");
}}}
```

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

```
class A
{}
class B
{}
class MyClass {
public static void main(String[] args)
{
Object o = class.forName(args[0]).newInstance();
System.out.println("Class created for" + o.getClass().getName());
}}
```

**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");
}}
}

```

**IOException :** It is thrown when an input-output operation failed or interrupted

```

import java.io.*;
class Sample {
public static void main(String args[])
{
FileInputStream f = null;
f = new FileInputStream("abc.txt");
int i;
while ((i = f.read()) != -1) {
System.out.print((char)i);
}
f.close();
}}

```

**InterruptedException :** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

```

class sample {
public static void main(String args[])
{
Thread t = new Thread();
t.sleep(10000);
}
}

```

**NoSuchMethodException :** It is thrown when accessing a method which is not found.

```

class Geeks
{
public Geeks()
{
Class i;
try {
i = Class.forName("java.lang.String");
try {
Class[] p = new Class[5];
}
catch (SecurityException e)
{
e.printStackTrace();
}
catch (NoSuchMethodException e)
{
e.printStackTrace();
}
}
}
}

```

```

}}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}}
public static void main(String[] args)
{
new Geeks();
}}

```

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

```

class NullPointerException_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..");
}}
}

```

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

```

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

```

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

```

class StringIndexOutOfBoundsException_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");
}}
}

```

### Creating Own Exceptions in Java :

Java allows the user to create their own exception class which is derived from built-in class Exception. The Exception class inherits all the methods from the class Throwable. The Throwable class is the superclass of all errors and exceptions in the Java language. It contains a

snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error.

- The Exception class is defined in java.lang package.
- User defined exception class must inherit Exception class.
- The user defined exception can be thrown using throw keyword.

**Syntax:**

```
class User_defined_name extends Exception
{
    .....
}
```

*Some of the methods defined by Throwable are shown in below table.*

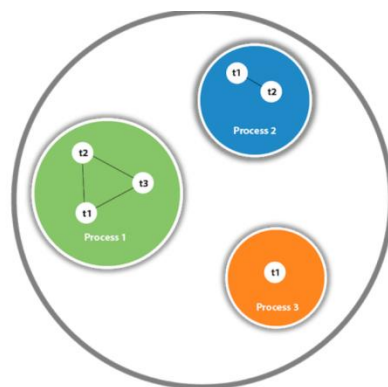
Methods	Description
Throwable fillInStackTrace()	Fills in the execution stack trace and returns a Throwable object.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
void printStackTrace()	Displays the stack trace.
String toString()	Returns a String object containing a description of the Exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement.

**Two commonly used constructors of Exception class are:**

- Exception() - Constructs a new exception with null as its detail message.
- Exception(String message) - Constructs a new exception with the specified detail message.

**THREAD**

Thread is a lightweight process, the 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 uses a shared memory area.



**Life cycle of a Thread (Thread States)**

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

- New Born state
- Runnable (Ready)
- Running

- Blocked(Waiting)
- Terminated(Dead)

#### **New Born state:**

When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it's code is yet to be run and hasn't started to execute.

#### **Runnable State:**

A thread that is ready to run is moved to runnable state.

In this state, a thread might actually be running or it might be ready run at any instant of time.

It is the responsibility of the thread scheduler to give the thread, time to run.

A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

#### **Blocked/Waiting state:**

When a thread is temporarily inactive, then it's in one of the following states:

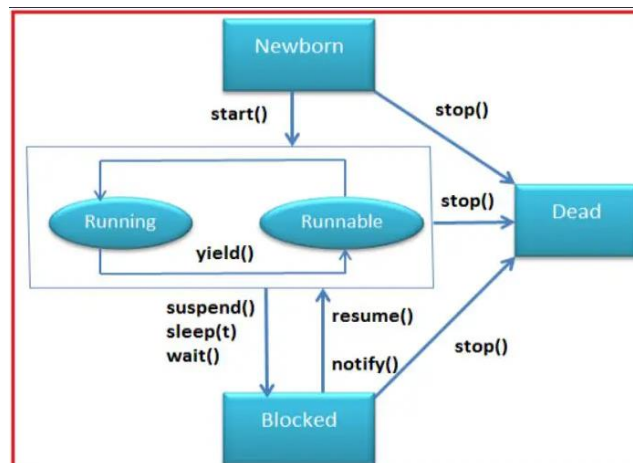
- Blocked
- Waiting

#### **Terminated(Dead)**

A thread is in terminated or dead state when its run() method exits. Because it exits normally. This happens when the code of thread has entirely executed by the program.

Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

A thread that lies in terminated state does no longer consumes any cycles of CPU.



#### **Commonly used methods of Thread class:**

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

public void start(): starts the execution of the thread. JVM calls the run() method on the thread.

public void sleep(long milliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void join(): waits for a thread to die.

public void join(long milliseconds): waits for a thread to die for the specified milliseconds



public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the priority of the thread.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

public Thread currentThread(): returns the reference of currently executing thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

public boolean isAlive(): tests if the thread is alive

public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

public void suspend(): is used to suspend the thread(deprecated)

public void resume(): is used to resume the suspended thread(deprecated)

public void stop(): is used to stop the thread(deprecated).

public boolean isDaemon(): tests if the thread is a daemon thread

public void setDaemon(boolean b): marks the thread as daemon or user thread.

public void interrupt(): interrupts the thread.

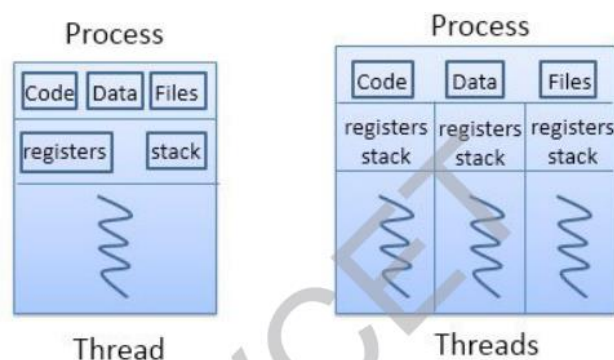
public boolean isInterrupted(): tests if the thread has been interrupted.

public static boolean interrupted(): tests if the current thread has been interrupted

## MULTITHREADING:

Multithreading is different from multitasking in a sense that multitasking allows multiple tasks at the same time, whereas, the Multithreading allows multiple threads of a single task (program, process) to be processed by CPU at the same time.

A thread is a basic execution unit which has its own program counter, set of the register and stack. But it shares the code, data, and file of the process to which it belongs. A process can have multiple threads simultaneously, and the CPU switches among these threads so frequently making an impression on the user that all threads are running simultaneously.



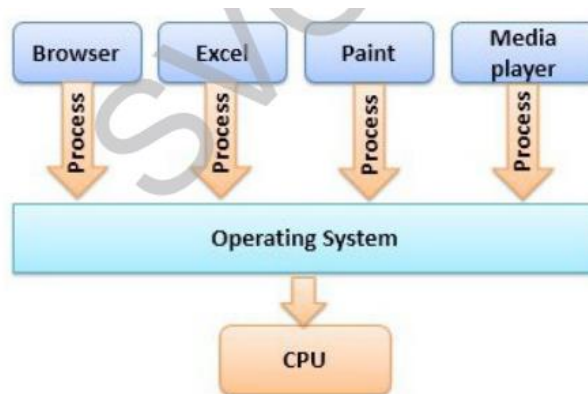
## Advantage of the Multithreading

- It enables you to write very efficient programs that maximizes the CPU utilization and reduces the idle time.
- Most I/O devices such as network ports, disk drives or the keyboard are much slower than CPU
- A program will spend much of its time just send and receive the information to or from the devices, which in turn wastes the CPU valuable time.
- By using the multithreading, your program can perform another task during this idle time.
- For example, while one part of the program is sending a file over the internet, another part can read the input from the keyboard, while other part can buffer the next block to send.
- It is possible to run two or more threads in multiprocessor or multi core systems simultaneously.

## MULTI-TASKING

Multitasking is when a single CPU performs several tasks (program, process, task, threads) at the same time. To perform multitasking, the CPU switches among these tasks very frequently so that user can interact with each program simultaneously

In a multitasking operating system, several users can share the system simultaneously. CPU rapidly switches among the tasks, so a little time is needed to switch from one user to the next user. This puts an impression on a user that entire computer system is dedicated to him.



*Figure: Multitasking*

When several users are sharing a multitasking operating system, CPU scheduling and multiprogramming makes it possible for each user to have at least a small portion of Multitasking OS and let each user have at least one program in the memory for execution.

### Types of multitasking:

- Process based Multitasking
- Thread based Multitasking

### Process based Multitasking:

Executing various jobs together where each job is a separate independent operation is called process-based multitasking. It is best suitable for the operating system level.

### Thread based Multitasking:

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread based multitasking and each independent part is called Thread.

It is best suitable for the programmatic level.

The main goal of multitasking is to make or do a better performance of the system by reducing response time.

### Comparison between multithreading and multi-tasking

MULTI TASKING	MULTI THREADING
Multitasking is to run multiple processes on a computer concurrently.	Multithreading is to execute multiple threads in a process concurrently.
In Multitasking, the CPU switches between multiple processes to complete the execution.	In Multithreading, the CPU switches between multiple threads in the same process
In Multitasking, resources are shared among multiple processes	In Multithreading, resources are shared among multiple threads in a process
Multitasking is heavy-weight and harder to create	Multithreading is light-weight and easy to create.

PROCESS BASED MULTITASKING	THREAD BASED MULTITASKING
Two or more process/program can be run concurrently.	Two or more threads can be run concurrently.
The smallest unit of execution is a process	The smallest unit of execution is a thread
Inter process communication is costly and inefficient.	Inter process communication is inexpensive and efficient
Processes take more time for context switching	Threads takes less time for context switching
It is unable to gain access over CPU idle time	It can gain access over CPU idle time
Every program has its own address space	Thread share the same address space
It has slower data rate multitasking	It has faster data rate multithreading or tasting

### CREATING A THREAD:

A thread is a lightweight process. Every java program executes by a thread called the main thread. When a java program gets executed, the main thread created automatically. All other threads called from the main thread. The java programming language provides two methods to create threads, and they are listed below.

- by extending Thread class
- by implementing Runnable interface

#### Extending Thread class:

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

- **Step-1:** Create a class as a child of Thread class. That means, create a class that extends Thread class.

#### Syntax:

```
Class udef extends Thread
{
    // statements
Public void run ( )
{
    // code for thread
}
// statements
```

- **Step-2:** Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

**Syntax:**

```
Public void run ( )
{
    // thread code here
}
```

- **Step-3:** Create the object of the newly created class in the main( ) method.

**Syntax:**

```
    udef thobjname = new udef( );
    thobjname.start ( );
```

- **Step-4:** Call the start( ) method on the object created in the above step.

Example program:

```
import java.io.*;
import java.lang.*;
class A extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("From Threaad A :i="+i);
        }
        System.out.println("Exit from Thread A");
    }
}
class B extends Thread
{
    public void run()
    {
        for(int j=1;j<=5;j++)
        {
            System.out.println("From Threaad B :j="+j);
        }
        System.out.println("Exit from Thread B");
    }
}
class C extends Thread
{
    public void run()
    {
        for(int k=1;k<=5;k++)
        {
            System.out.println("From Threaad C :k="+k);
        }
        System.out.println("Exit from Thread C");
    }
}
class ThreadTest
{
    public static void main(String args[])
    {
        System.out.println("main thread started");
        A a=new A();
        a.start();
```

```

B b=new B();
b.start();
C c=new C();
c.start();
System.out.println("main thread ended");
}}

```

Output:

```

main thread started
From Threaad A :i=1
From Threaad A :i=2
From Threaad A :i=3
From Threaad A :i=4
From Threaad A :i=5
Exit from Thread A
main thread ended
From Threaad C :k=1
From Threaad C :k=2
From Threaad C :k=3
From Threaad C :k=4
From Threaad C :k=5
Exit from Thread C
From Threaad B :j=1
From Threaad B :j=2
From Threaad B :j=3
From Threaad B :j=4
From Threaad B :j=5
Exit from Thread B

```

### Implementng Runnable interface

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

- **Step-1:** Create a class that implements Runnable interface.

#### Syntax:

```

Class udef implements Runnable
{
    // statements
Public void run ( )
{
    // code for thread
}
    // statements
}

```

- **Step-2:** Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

#### Syntax:

```

Public void run ( )
{
    // thread code here
}

```

- **Step-3:** Create the object of the newly created class in the main( ) method.
- **Step-4:** Create the Thread class object by passing above created object as parameter to the Thread class constructor.

#### Syntax:

```

    udef testx = new udef( );
    Thread testthread = new Thread (testx);
    Thestthread.start ( );

```

- **Step-5:** Call the start() method on the Thread class object created in the above step.

### Stopping the thread

Whenever we want to stop a thread from running state by calling **stop()** method of **Thread** class in Java.

### Syntax

```
Threadobject.stop ( );
```

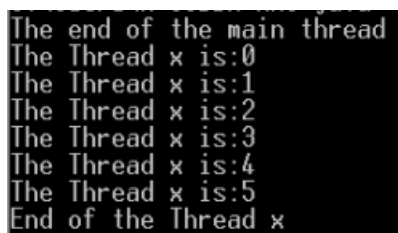
Example program:

```

class x implements Runnable
{
    public void run()
    {
        for(int i=0;i<=5;i++)
            System.out.println("The Thread x is:"+i);
        System.out.println("End of the Thread x");
    }
}
class RunnableTest
{
    public static void main(String args[])
    {
        x r=new x();
        Thread threadx=new Thread(r);
        threadx.start();
        System.out.println("The end of the main thread");
    }
}

```

Output:



```

The end of the main thread
The Thread x is:0
The Thread x is:1
The Thread x is:2
The Thread x is:3
The Thread x is:4
The Thread x is:5
End of the Thread x

```

### THREAD PRIORITY:

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence.

To set a thread's priority, use the setPriority() method, which is a member of Thread.

### Syntax:

```
Threadname.setpriority(int number (or) priority constants);
```

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

**MAX\_PRIORITY** - It has the value 10 and indicates highest priority.

**NORM\_PRIORITY** - It has the value 5 and indicates normal priority.

**MIN\_PRIORITY** - It has the value 1 and indicates lowest priority.

The default priority of any thread is 5 (i.e. NORM\_PRIORITY).

### Example

```
threadObject.setPriority(4);
```

or

```
threadObject.setPriority(MAX_PRIORITY);
```

### Example Program:

```
class PThread1 extends Thread
{
public void run()
{
System.out.println(" Child 1 is started");
}}
class PThread2 extends Thread
{
public void run()
{
System.out.println(" Child 2 is started");
}}
class PThread3 extends Thread
{
public void run()
{
System.out.println(" Child 3 is started");
}}
class PTest
{
public static void main(String args[])
{
PThread1 pt1=new PThread1();
pt1.setPriority(1); PThread2
pt2=new PThread2();
pt2.setPriority(9);
PThread3 pt3=new PThread3();
pt3.setPriority(6);
pt1.start();
pt2.start();
pt3.start();
System.out.println("The pt1 thread priority is :"+pt1.getPriority());
}}
```

### Thread Synchronization:

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

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

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

Synchronization is mainly used

To prevent thread interference

To prevent consistency problem

### Types:

There are 2 types of thread synchronization

- ❖ Mutual exclusive
- ❖ Inter thread communication

### Using Synchronized Methods:

- If u declare any method as synchronized it is known as synchronized method.
- Synchronization is implemented by internal entity known as Lock or monitor.
- To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
- Key to synchronization is the concept of the monitor (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock, or mutex.
- Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor

### Example using the synchronized method

Class Table

```
{
synchronized void printTable(int n)
{
for(int i=1;i<=5;i++)
{
System.out.println(n*i);
try{
Thread.sleep(400);
}
catch(InterruptedException ie)
System.out.println("The Exception is :"+ie);
}}}}
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 TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();
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

```

## INTER-THREAD COMMUNICATION

Inter-process communication (IPC) is a mechanism that allows the exchange of data between processes. If two or more Threads are communicating with each other, it is called "inter thread" communication. Using the synchronized method, two or more threads can communicate indirectly. Through, synchronized method, each thread always competes for the resource. This way of competing is called polling.

Java addresses this polling problem, using via **wait()**, **notify()**, and **notifyAll()** methods.

**wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()**.

**notify()** wakes up a thread that called **wait()** on the same object.

**notifyAll()** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

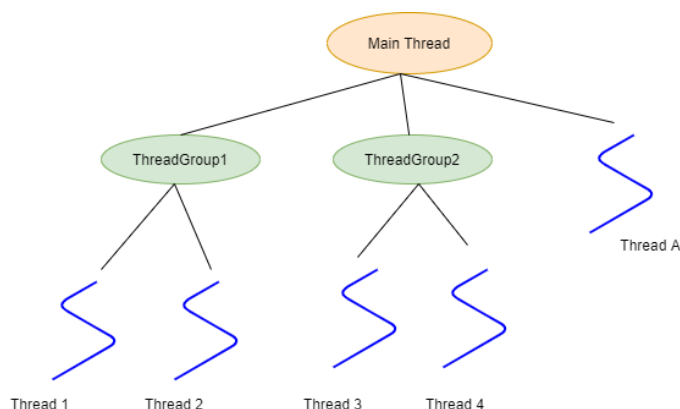
## THREAD GROUP IN JAVA

Java thread group is implemented by `java.lang.ThreadGroup` class.

A `ThreadGroup` represents a set of threads.

A thread group is a way to group multiple threads together. It is represented by the `ThreadGroup` class in the `java.lang` package.

This group of threads are in the form of a tree structure, in which the initial thread is the parent thread.



```

public class ThreadGroupDemo implements Runnable{
public void run() {
System.out.println(Thread.currentThread().getName());
}
}

```

```

public static void main(String[] args) {
    ThreadGroupDemo runnable = new ThreadGroupDemo();
    ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
    Thread t1 = new Thread(tg1, runnable, "one");
    t1.start();
    Thread t2 = new Thread(tg1, runnable, "two");
    t2.start();
    Thread t3 = new Thread(tg1, runnable, "three");
    t3.start();
    System.out.println("Thread Group Name: "+tg1.getName());
    tg1.list();

}
}

```

## Daemon Thread in Java

A daemon thread is a special kind of thread that runs in the background and does not prevent the Java Virtual Machine (JVM) from exiting when all non-daemon threads have completed their execution.

Daemon thread in Java is a low-priority thread that performs background operations such as garbage collection, finalizer, Action Listeners, Signal dispatches, etc.

## Methods for Java Daemon thread

- public void setDaemon(boolean status)
- public boolean isDaemon()

Daemon threads are typically categorized into two main types based on their purpose and behavior:

- System Daemon Threads
- User Daemon Threads

## JAVA ANNOTATIONS

Java annotations are metadata (data about data) for our program source code.

They provide additional information about the program to the compiler but are not part of the program itself. These annotations do not affect the execution of the compiled program. i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

## Built-In Java Annotations

There are several built-in annotations in Java. Some annotations are applied to Java code and some to other annotations.

@Override annotation.  
 @ displayInfo()  
 @FunctionalInterfac

```

class Animal {
    public void displayInfo() {
        System.out.println("I am an animal.");
    }
}
class Dog extends Animal {
    @Override

```

```

public void displayInfo() {
    System.out.println("I am a dog.");
}
}
class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}

```

## ENUMERATION

- Enumerations was added to Java language in JDK5.
- Enumeration means a list of named constant.
- In Java, enumeration defines a class type.
- An Enumeration can have constructors, methods and instance variables.
- It is created using enum keyword.
- Each enumeration constant is public, static and final by default.
- Even though enumeration defines a class type and have constructors, you do not instantiate an enum using new.
- Enumeration variables are used and declared in much a same way as you do a primitive variable.

### Syntax for defining Enumeration:

```
enum enumerationtype {identifier, identifier2, .....}
```

### Example:

```

enum Season { WINTER, SPRING, SUMMER, FALL }
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY }

```

### Syntax for declaring Enumeration variable:

```
enumerationtype variablename1, variablename2, .....;
```

### Example:

```

Season s;
Day d;

```

enumeration variable can accept only enumeration constant as its value.

```

s= Season. WINTER;
d= Day. SUNDAY

```

### Initializing specific value to the enum constants:

The enum constants have initial value that starts from 0, 1, 2, 3 and so on. But we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors and methods.

```
enum Season { WINTER(5), SPRING(10), SUMMER(15), FALL(20);
```

### Methods of Enumeration

All enumerations automatically contain predefined methods: values( ) valueOf( ) and ordinal() Their general forms are shown here:

```

public static enum-type[ ] values( );
public static enum-type valueOf(String str);
public final int ordinal();

```

The values( ) method returns an array that contains a list of the enumeration constants.

The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.

The ordinal() method returns this enumeration constant's ordinal.

**The following program demonstrates the values(), valueOf() and ordinal() methods:**

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumDemo2 {
public static void main(String args[])
{
Season s;
Season allseasons[] = Season.values(); // use values()
for(Season x : allseasons)
System.out.println(x);
System.out.println();
s = Season.valueOf("SUMMER"); // use valueOf()
System.out.println("s contains " + s);
System.out.println("WINTER ordinal="+Season.WINTER.ordinal());
System.out.println("SPRING ordinal="+Season.SPRING.ordinal());
System.out.println("SUMMER ordinal="+Season.SUMMER.ordinal());
System.out.println(" FALL ordinal="+Season.FALL.ordinal());
}}
```

**Output:**

```
WINTER
SPRING
SUMMER
FALL
s contains SUMMER
WINTER ordinal=0
SPRING ordinal=1
SUMMER ordinal=2
FALL ordinal=3
```

### **AUTOBOXING AND UNBOXING:**

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

#### **Advantage of Autoboxing and Unboxing:**

No need of conversion between primitives and Wrappers manually so less coding is required.

#### **Simple Example of Autoboxing in java:**

```
class BoxingExample1
{
public static void main(String args[])
{
int a=50;
Integer a2=new Integer(a); //Boxing
Integer a3=5; //Boxing
System.out.println(a2+" "+a3);
}}
```

**Output:50 5**

#### **Simple Example of Unboxing in java:**

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

```
class UnboxingExample1
{
```

```
public static void main(String args[])
{
Integer i=new Integer(50);
int a=i;
System.out.println(a);
}}
```

**Output: 50**

## **GENERIC**

Generics means parameterized types. The idea is to allow type (Integer, String, ... etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.

A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.

### **The General Form of a Generic Class**

```
class class-name <type-param-list>{ // ... }
```

### **Syntax for declaring a generic class:**

```
class-name<type-arg-list> var-name =new class-name<type-arg-list> (cons-arg-list)
```