Java : Unit III

Interface, Abstract, Packages and Exception Handling

Interface

- An interface in Java is a blueprint of a class. It has static constants and abstract methods.

- The interface in Java is *a* mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

- Since it does not have method body you cannot create an object for interface.

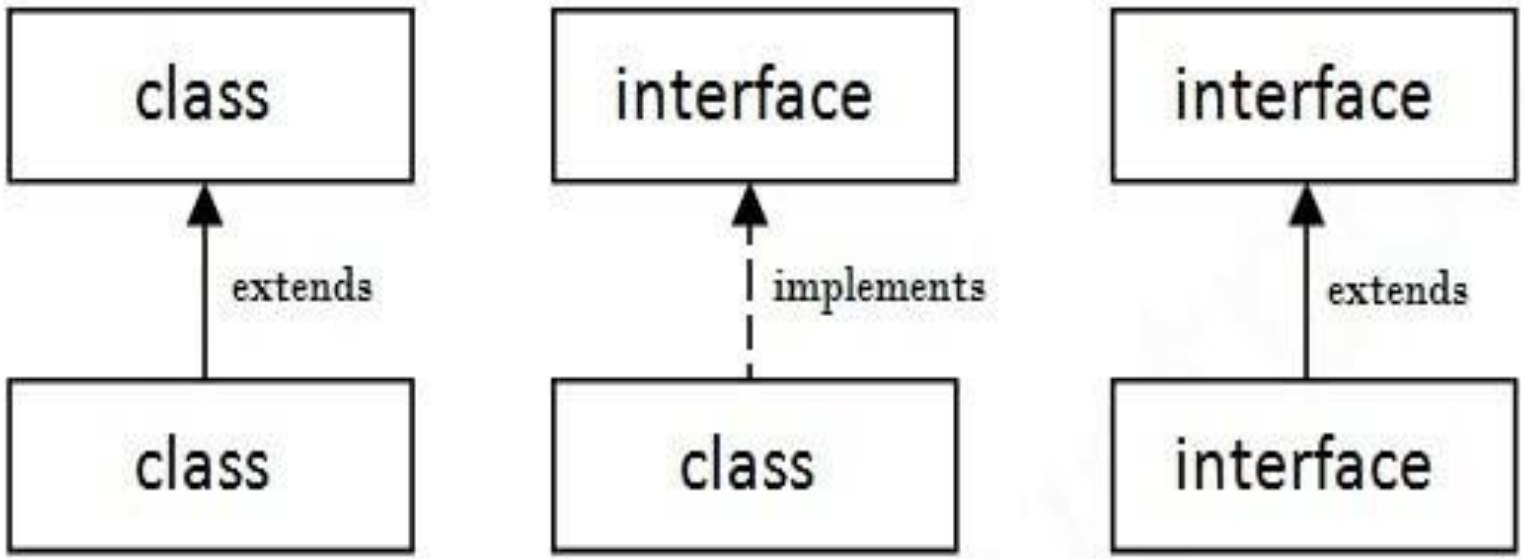
How to declare an interface?

- An interface is declared by using the interface keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.
- Syntax:

interface <interface_name>{

```
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

The relationship between classes and interfaces

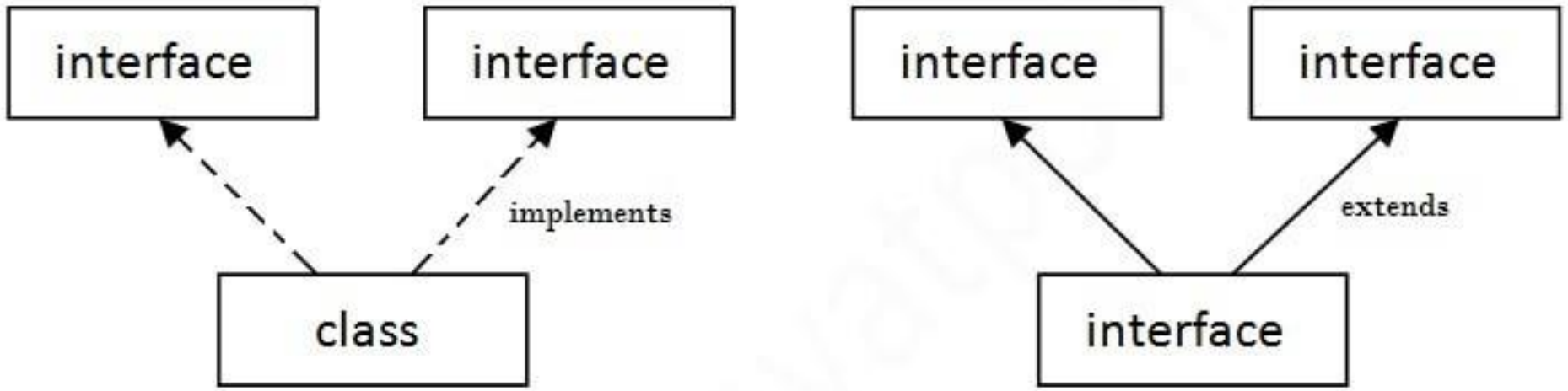


Java Interface Example

```
interface printable{
void print();
}
class test implements printable{
public void print()
{
        System.out.println(“Example of interface");
}
public static void main(String args[]){
test obj = new test();
obj.print();
 }
}
```

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Multiple inheritance Example

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class test implements Printable,Showable{
public void print(){System.out.println(“Message from first Interface");}
public void show(){System.out.println(“Message from second Interface");}

public static void main(String args[]){
test obj = new test();
System.out.println(“Example of multiple interface");
obj.print();
obj.show();
 }
}
```

Interface inheritance (Extending an interface)

```
interface Printable{
void print();
}
interface Showable extends Printable{                    // Extending an interface
void show();
}
class TestInterface implements Showable{
public void print(){System.out.println(“Print method from first interface");}
public void show(){System.out.println(“Show method from second interface");}

public static void main(String args[]){
TestInterface obj = new TestInterface();
System.out.println(“Example of extending an interface");
obj.print();
obj.show();
 }
}
```

Abstraction in Java

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

- Abstraction lets you focus on what the object does instead of how it does it.

- Ways to achieve Abstraction

  - Abstract class (0 to 100%)
  - Interface (100%)

Abstraction in Java

- A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

- Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of Abstract class that has an abstract method

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Abstract class having constructor, data member and methods

```
abstract class abs
{
        abs()
        {
                System.out.println("Constructoe in abstract class");
        }
        void normal()
        {
                System.out.println("Normal Method in abstract class");
        }
        abstract void print();
}
class abst extends abs
{
        void print()
        {
                System.out.println("Example of abstract class");
        }
        public static void main(String args[])
        {
                abst a=new abst();
                a.print();
                a.normal();
        }
```

Difference between Abstract class and Interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Packages in Java

- A **java package** is a group of similar types of classes, interfaces and sub-packages.

- Package in java can be categorized in two form, built-in package and user-defined package.

- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Packages in Java

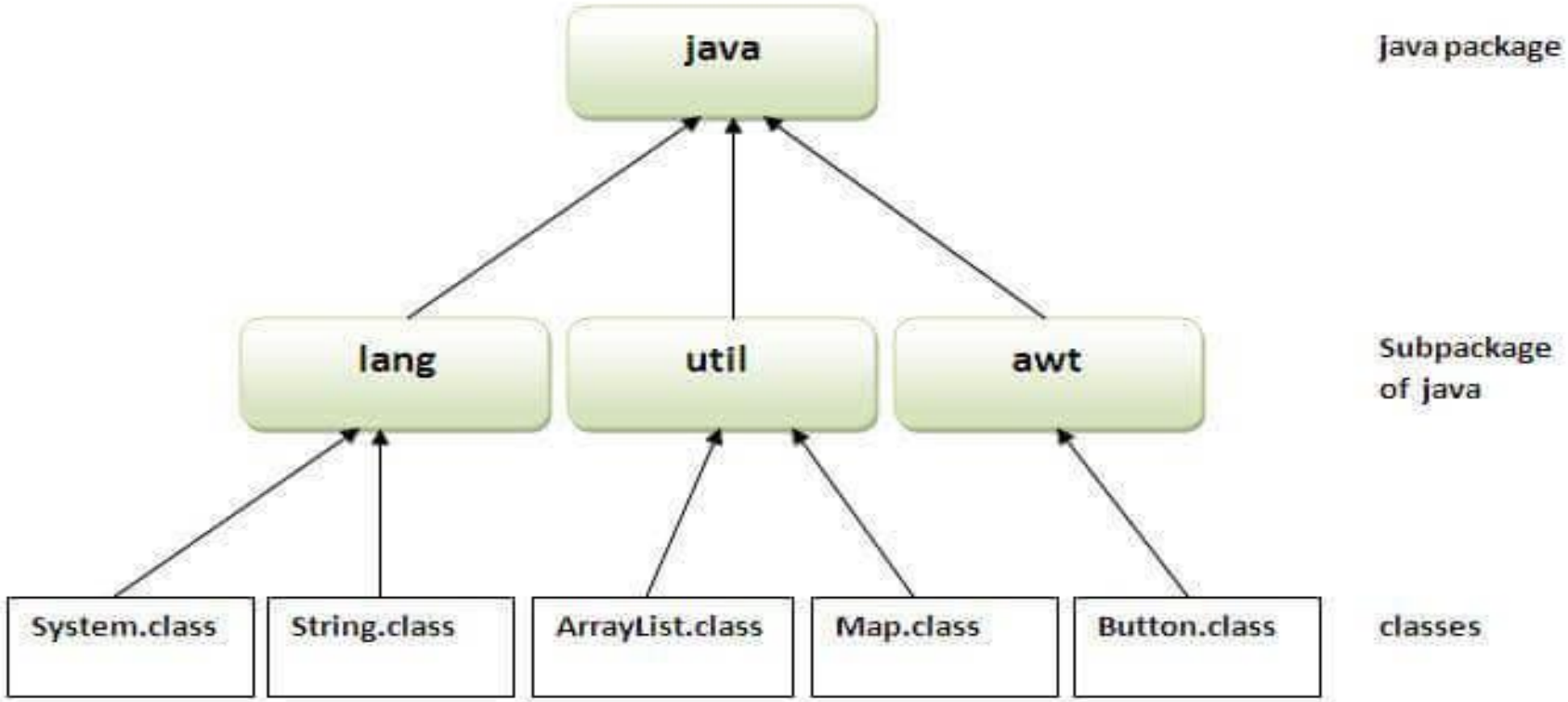
- Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Packages in Java



Simple example of java package

The **package keyword** is used to create a package in java.

//save as Simple.java

```
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

How to compile java package

- If you are not using any IDE, you need to follow the **syntax** given below:
- javac -d directory javafilename
- For **example**
- javac -d . Simple.java
- The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc.
- If you want to keep the package within the same directory, you can use . (dot).

Output:Welcome to package

How to run java package

You need to use fully qualified name e.g. mypack.Simple etc to run the class

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

Output:Welcome to Edureka

How to access package from another package?

- There are three ways to access the package from outside the package.

1. import package.*;

2. import package.classname;

3. Fully qualified name.

Output:Welcome to package

Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not sub packages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

Output:Welcome to package

Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output:Welcome to package

Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible.
- Now there is no need to import.
- But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Output:Welcome to package

Using fully qualified name

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

```
Output:Hello
```

Output:Welcome to package

Sub package in java

- Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Example of Subpackage

```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

```
Output:Hello subpackage
```

Output:Welcome to package

Exception Handling in Java

- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Output:Welcome to package

Exception Handling in Java

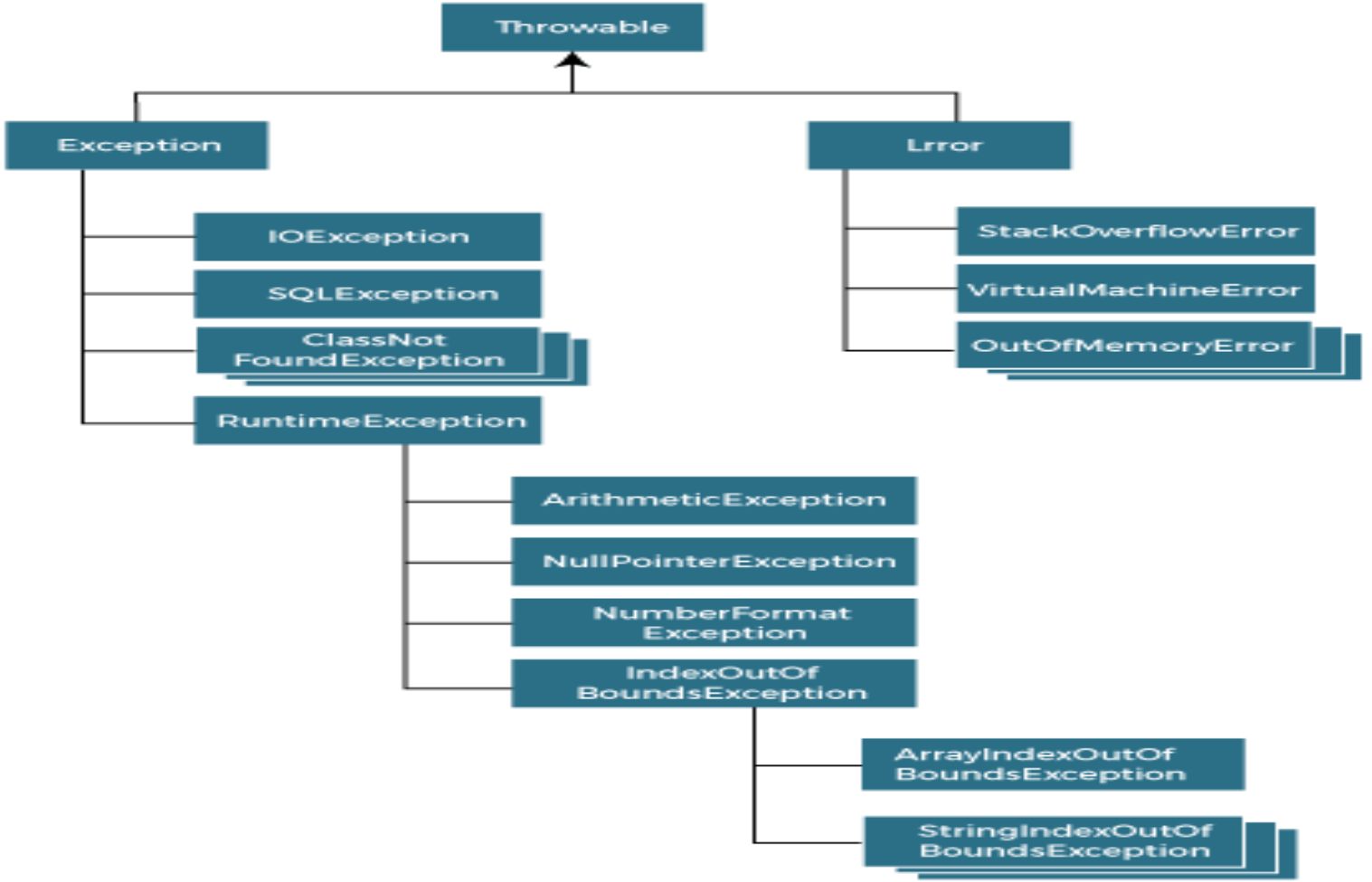
- Advantage of Exception Handling
- The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:
- statement 1;
- statement 2;
- statement 3;
- statement 4;
- statement 5;//exception occurs
- statement 6;
- statement 7;
- statement 8;
- statement 9;
- statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Output:Welcome to package

Hierarchy of Java Exception classes

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:



Output:Welcome to package

Types of Java Exceptions

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

- Checked Exception

- Unchecked Exception

- Error

Output:Welcome to package

Types of Java Exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Output:Welcome to package

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Output:Welcome to package

Common Scenarios of Java Exceptions

1) A scenario where ArithmeticException occurs

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

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

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

Output:Welcome to package

Common Scenarios of Java Exceptions

3. A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

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

Output:Welcome to package

Java Try Catch Block

```
public class TryCatchExample2 {

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

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Output:Welcome to package

Java Catch Multiple Exceptions

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

- Points to remember
- At a time only one exception occurs and at a time only one catch block is executed.

- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Output:Welcome to package

Java Catch Multiple Exceptions

```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
                System.out.println(a[10]);
}
            catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
            catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
            catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
            System.out.println("rest of the code");
    }
}
```

Output:
ArrayIndexOutOfBoundsException Occurs

Rest of the code

Output:Welcome to package

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 **ArithemeticException** (division by zero).

- 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.

Output:Welcome to package

Java Nested try block

```
class nestedtry
{
        public static void main(String args[])
        {
                try
                {
                        try
                        {
                                int data=100/0;
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                        try
                        {
                                int[] a={1,2,3,4,5};
                                System.out.println(a[10]);
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                }
                finally
                {
                        System.out.println("Example of nested try");
                }
        }
}
```

Output :
ArithmeticException : Divide by zero

ArrayIndexOutOfBoundsException

Example of nested try

Output:Welcome to package

Java throw keyword

- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.
- The syntax of the Java throw keyword is given below.
- throw Instance i.e.,
    - **throw new** exception_class("error message");
- Let's see the example of throw IOException.
    - **throw new** IOException("sorry device error");

Output:Welcome to package

Java throw keyword Example

```
public class TestThrow1 {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

Output:Welcome to package

Java 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 the normal flow of the program can be maintained.
- Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

• Syntax of Java throws

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

Output:Welcome to package

Java throws keyword Example

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

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

Output:

```
Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)
```

Output:Welcome to package

Java Throw v/s Throws

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	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.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Output:Welcome to package

Difference between final, finally and finalize

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.
3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.