UNIT-IV

INHERITANCE AND EXCEPTION HANDLING

Inheritance in Java

- Relationship in java

A combination of two or more classes in java is known as a relationship.

In Java, we have two types of relationship:

1. **Is-A relationship - Inheritance**
2. **Has-A relationship - Aggregation**

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs

When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

class Subclass-name **extends** Superclass-name

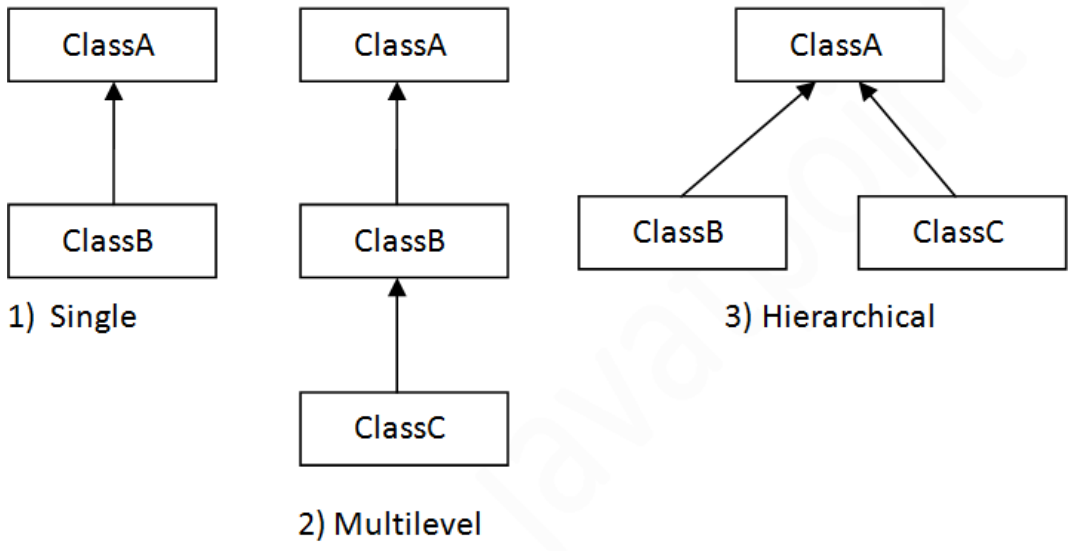
```
1. {
2.   //methods and fields
3. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

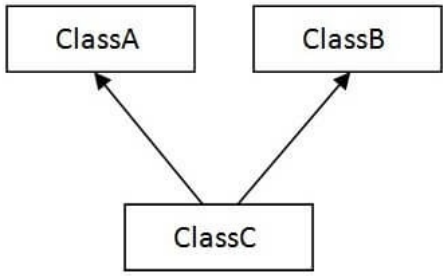
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

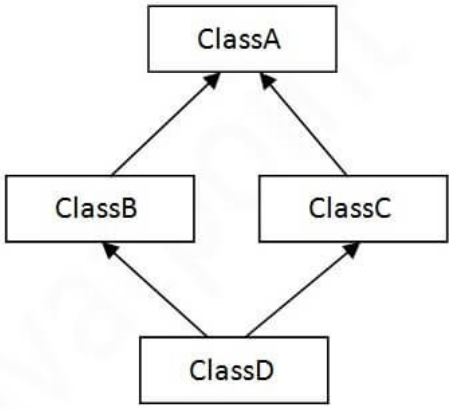
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.



4) Multiple



5) Hybrid

.

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*.

```
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

class one {
    public void geek()
    {
        System.out.println("Geeks");
    }
}

class two extends one
{
    public void forever()
```

```
    {
    System.out.println("for");
    }
}
// Driver class
public class Main
{
    public static void main(String[] args)
    {
        two g = new two();
        g.geek();
        g.forever();
        g.geek();
    }
}
```

Output

Geeks

for

Geeks

Multilevel Inheritance

Example

In this example we have three classes – `Car`, `Maruti` and `Maruti800`. We have done a setup – class Maruti extends `Car` and class `Maruti800` extends `Maruti`. With the help of this Multilevel hierarchy setup our Maruti800 class is able to use the methods of both the classes (Car and Maruti).

```
class Car{
   public Car()
   {
       System.out.println("Class Car");
   }
   public void vehicleType()
   {
       System.out.println("Vehicle Type: Car");
```

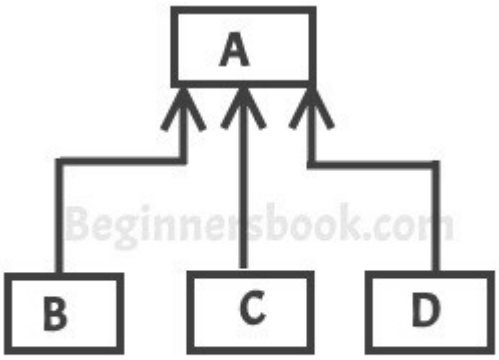
```
    }
}
class Maruti extends Car{
    public Maruti()
    {
        System.out.println("Class Maruti");
    }
    public void brand()
    {
        System.out.println("Brand: Maruti");
    }
    public void speed()
    {
        System.out.println("Max: 90Kmph");
    }
}
public class Maruti800 extends Maruti{

    public Maruti800()
    {
        System.out.println("Maruti Model: 800");
    }
    public void speed()
    {
        System.out.println("Max: 80Kmph");
    }
    public static void main(String args[])
    {
         Maruti800 obj=new
         Maruti800();
         obj.vehicleType();
         obj.brand();
         obj.speed();
    }
```

Output:

```
Class Car
Class Maruti
Maruti Model: 800
Vehicle Type: Car
Brand: Maruti
Max: 80Kmph
```

Hierarchical Inheritance



Hierarchical Inheritance

We are writing the program where class B, C and D extends class A.

```
class A
{
   public void methodA()
   {
      System.out.println("method of Class A");
   }
}
class B extends A
{
   public void methodB()
   {
      System.out.println("method of Class B");
   }
}
class C extends A
{
  public void methodC()
  {
     System.out.println("method of Class C");
  }
}
class D extends A
{
  public void methodD()
  {
     System.out.println("method of Class D");
  }
}
class JavaExample
{
  public static void main(String args[])
  {
```

```
        B obj1 = new B();
        C obj2 = new C();
        D obj3 = new D();
        //All classes can access the method of class A
        obj1.methodA();
        obj2.methodA();
        obj3.methodA();
        obj1.methodB();
        obj2.methodC();
        obj3.methodD();
    }
}
```

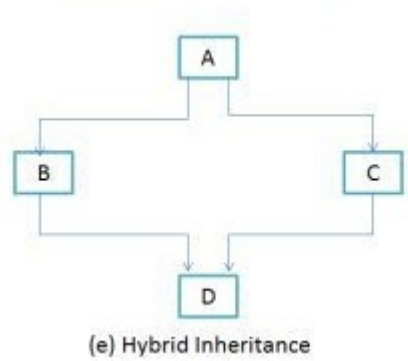
Output:

```
method of Class A
method of Class A
method of Class A
```

Exercise: Write a program to create a player class. Inherit the classes Cricket_player, Football_player and Hockey_player form playerclass.

Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



(e) Hybrid Inheritance

Public class c extends a extends b
Implications of inheritance:

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A extends B{  //not permited in java
8.
9.  public static void main(String args[]){
10.   C obj=new C();
11.   obj.msg();//Now which msg() method would be invoked?
12. }
13. }
```

```
Compile Time Error
```

Interface in java

What is an interface in Java?

- Interface looks like a class but it is not a class.

- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body).
- Also, the variables declared in an interface are public, static & final by default.
- Since methods in interfaces do not have body, they have to be implemented by the class before you can access them.
- The class that implements interface must implement all the methods of that interface.
- Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

Why do we use an Interface?

- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface.

Following is an syntax of an interface −

```
import java.lang.*;
// Any number of import statements

public interface NameOfInterface
{
   // Any number of final, static fields- variable declaration
```

```
  // Any number of abstract method declarations only
}
```

Example:

```
// A simple interface
interface Player
{
    static final int id = 10; String
    name="B.Sc";
      void move(); int
        give();
}
```

Syntax to implement interface in class:

```
interface interfacename
{
    //variable and method declaration;
}
```

1. class classname implements interfacename
2. {
3. //class definition;
4. //method definition from interface
   }

Java Interface Example: Bank

5. import java.io.*;
6. import java.util.*;

```
7. import java.lang.*;
8. interface Bank
```

```
{
float  r=9.15;
void rateOfInterest( );
}
class CRDB implements Bank
{
public void rateOfInterest()
{
System.out.println(r);
}
}
class bankinter
{
public static void main(String[ ] args)
{
CRDB c=new CRDB( );
System.out.println("ROI:CRDB  ");
c.rateOfInterest( );
}}
```

Output:

```
ROI:CRDB 9.15
```

Java Interface Example: A class implements two interfaces (multiple inheritance)

```
import java.io.*;
import
java.util.*;
import
java.lang.*;

interface Sing
{
String s=”Irene”;
 void singRock();
}
```

```
void danceSlow(String s2);
}

public class Human implements Sing, Dance
{
void singRock()
{
System.out.println(s+”  “ +“I am singing rock”);
}
void danceSlow(String s2)
{
System.out.println(s2+”  “ +“I am dancing slow!”);
}
}


public class Talent
{
public static void main(String[] args)
{
Human h = new Human();
h.singRock();
h.danceSlow(“Hilary”);
}
}
```

Output:

Irene I am singing rock

Hilary I am dancing slow!

Example:2 **Java Interface Example: A class implements two interfaces (multiple inheritance)**

```
interface MotorBike
{
int speed=50;
public void totalDistance();
}

interface Cycle
{
int distance=150;
public void speed();
}

public class TwoWheeler implements MotorBike,Cycle
{

int totalDistance;
int avgSpeed;

public void totalDistance()
{
totalDistance=speed*distance;
System.out.println("Total Distance Travelled : "+totalDistance);
}

public void speed()
{
int avgSpeed=totalDistance/speed;
System.out.println("Average Speed maintained : "+avgSpeed);
}

public static void main(String args[])
{
TwoWheeler t1=new
TwoWheeler();
t1.totalDistance();
t1.speed();
}
```

Output

Total Distance Travelled: 7500

Average Speed maintained: 150

To achieve multiple inheritance in Java, we must use the interface.

Another Example: Multiple Inheritance in Java- A class extends one class and implements one interface

```
interface Backend {

   // abstract class
   public void connectServer();
}

class Frontend {

   public void responsive(String str)
  {
      System.out.println(str + " can also be used as frontend.");
   }
}

// Language extends Frontend class
// Language implements Backend interface
class Language extends Frontend implements Backend {

   String La = "Java";

   // implement method of interface public
   void connectServer() {
      System.out.println(language + " can be used as backend language.");
   }

   public static void main(String[] args) {

      // create object of Language class
      Language java = new Language();

      java.connectServer();

      // call the inherited method of Frontend class
      java.responsive(java.La);
   }
```

```
}
```

Output

```
Java can be used as backend language. Java
can also be used as frontend.
```

mpile Time Error
Compile Time Error

Aggregation in Java **(Has-A Relationship)**

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

```
Student Has-A Address (Has-a relationship between student and address)
College Has-A Address (Has-a relationship between college and address)
```

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

```
1. class Employee{
2. int id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.

- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

```
1.  public class Address {
2.  String city,state,country;
3.  public Address(String city, String state, String country) {
4.      this.city = city;
5.      this.state = state;
6.      this.country = country;
7.  }
8.
9.  }
1.  public class Emp {
2.  int id;
3.  String name;
4.  Address adrs;
5.
6.  public Emp(int id, String name,Address adrs) {
7.      this.id = id;
8.      this.name = name;
9.      this.adrs=adrs;
10. }
11.
12. void display( )
13. {
14. System.out.println(id+" "+name);
15. System.out.println(adrs.city+" "+adrs.state+" "+adrs.country);
16. }
17.
18. public static void main(String[] args) {
19. Address a1=new Address("gzb","Mbezi","tanzania");
```

```
20. Address a2=new Address("gno","Mbezi","tanzania");
22. Emp e=new Emp(111,"Smith",a1);
23. Emp e2=new Emp(112,"Henry",a2); 24.
25. e.display();
26. e2.display();
27.
28. }
29. }
```

```
Output:111 Smith
       gzb Mbezi tanzania
       112 Henry
       gno Mbezi tanzania
```

Garbage Collection in Java

Java garbage collection is the **process of releasing unused memory** occupied by unused objects. This process is done by the JVM automatically because it is essential for memory management.

When a Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed.

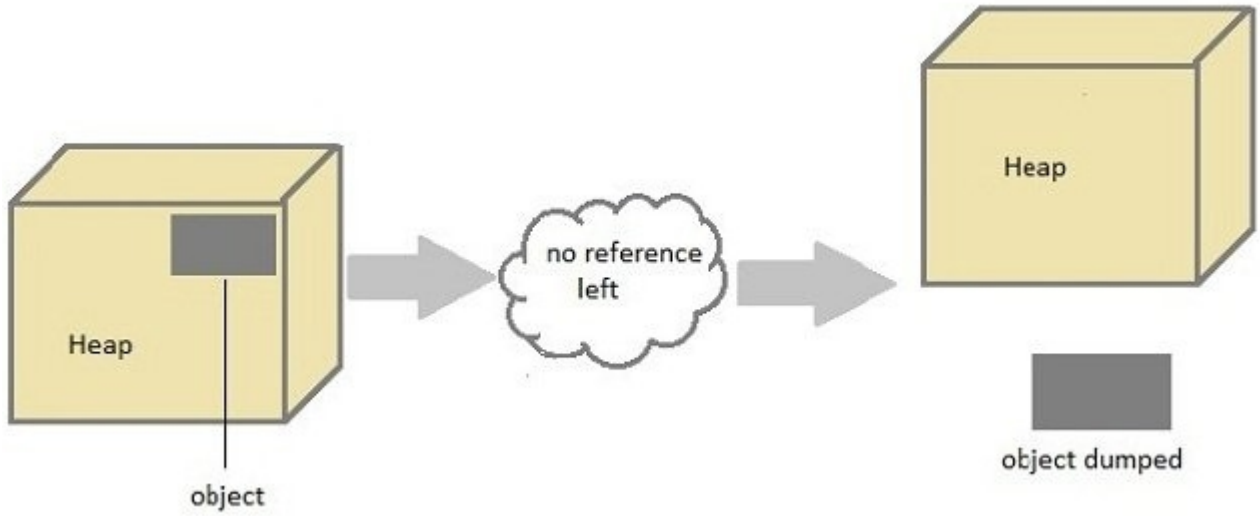
When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called **Garbage Collection**.

How Garbage Collection Works?

The garbage collection is a part of the JVM and is an automatic process done by JVM. We do not need to explicitly mark objects to be deleted. However, we can request to the JVM for garbage collection of an object but ultimately it depends on the JVM to call garbage collector.

Unlike C++ there is no explicit way to destroy object.

In the below image, you can understand that if an object does not have any reference than it will be dumped by the JVM.



Can the Garbage Collection be forced explicitly ?

No, the Garbage Collection cannot be forced explicitly. We may request JVM for **garbage collection** bycalling **System.gc()** method.

But this does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

An object is able to get garbage collect if it is non-reference. We can make an object non-reference by using three ways.

1. set null to object reference which makes it able for garbage collection. For example:

```
Demo demo = new Demo();

demo = null; // ready for garbage collection
```

2. We can non-reference an object by setting new reference to it which makes it able for garbage collection. For example

```
Demo demo = new Demo();

Demo demo2 = new Demo();

demo2 = demo // referring object
```

3. **Anonymous object** does not have any reference so if it is not in use, it is ready for the garbage collection.

---

finalize() method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation **finalize()** method is used.

The **finalize()** method is **called by garbage collection thread before collecting object**. It's the last chance for any object to perform cleanup utility.

Signature of **finalize()** method

```
protected void finalize()

{

 //finalize-code

}
```

Some Important Points to Remember

1. It is defined in java.lang.Object class, therefore it is available to all the classes.
2. It is declare as proctected inside Object class.
3. It gets called only once by a Daemon thread named GC (Garbage Collector) thread.

Request for Garbage Collection

We can request to JVM for garbage collection however, it is upto the JVM when to start the garbage collector.

Java **gc()** method is used to call garbage collector explicitly. However gc() method does not guarantee that JVM will perform the garbage collection. It only request the JVM for garbage collection. This method is present in **System and Runtime class**.

Example of gc() Method

Let's take an example and understand the functioning of the gc() method.

```
public class Test
```

```
{


    public static void main(String[] args)
    {
        Test t = new
        Test(); t=null;
        System.gc();
    }
    public void finalize()
    {
        System.out.println("Garbage Collected");
    }
}
```

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

```
1. String s="welcome";
```

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

```
1. String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```
1. import java.io.*;
2. import java.lang.*;
3. import java.util.*;
4. public class StringExample
5. {
6. public static void main(String args[])
7. {
8. String s1="java";//creating string by Java string literal
9. char ch[]={'s','t','r','i','n','g','s'};
10. String s2=new String(ch);  //converting char array to string
11. String s3=new String("example");//creating Java string by new keyword
12. System.out.println(s1);
13. System.out.println(s2);
14. System.out.println(s3);
15. }}
```

Output:

```
ja
strings
example
```

Java String class methods

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
4	boolean equals(Object another)	It checks the equality of string with the given object.
5	boolean isEmpty()	It checks if string is empty.
6	String concat(String str)	It concatenates the specified string.
7	String replace(char old, char new)	It replaces all occurrences of the specified char value.
8	int indexOf(int ch)	It returns the specified char value index.
9	int indexOf(String substring)	It returns the specified substring index.
10	String toLowerCase()	It returns a string in lowercase.
11	String toUpperCase()	It returns a string in uppercase.
12	String trim()	It removes beginning and ending spaces of this string.

1. import java.io.*;

```
2.  import java.lang.*;
3.  import java.util.*;
4.  class Teststringcomparison1{
5.   public static void main(String args[]){
6.     String s1="Ronaldo";
7.     String s2=new String("RONALDO");
8.      String s3="Messi";
1.     System.out.println(s1.equals(s2));//false
2.     System.out.println(s1.equalsIgnoreCase(s2));//true
3.  System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )

     String s4=s1.concat(s3);
       System.out.println(s4);
    String s="ChristianRonaldo";
    System.out.println("Original String: " + s);
     System.out.println("Substring starting from index 6: " +s.substring(6));//ianRonaldo
    System.out.println("Substring starting from index 0 to 6: "+s.substring(0,5)); //Christ
    String s6="Sachin";
    System.out.println(s6.toUpperCase());//SACHIN
    System.out.println(s6.toLowerCase());//sachin

    String s7="  Sachin  ";
    System.out.println(s7.trim());
    String s8="Messi";
    System.out.println(s.charAt(0));//M
    System.out.println(s8.charAt(3));//s
    System.out.println(s8.length());//5




9.   }
10. }
```

Output:

```
false
```

```
true
-1
RonaldoMessi
Original String: SachinTendulkar
Substring starting from index 6: ianRonaldo
Substring starting from index 0 to 6: Christ
SACHIN
sachin
M
s
5
```

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

append(String s)	It is used to append the specified string with this string.
capacity()	the total allocated capacity can be found by the capacity( ) method
insert(int offset, String s)	It is used to insert the specified string with this string at the specified position.
reverse()	is used to reverse the string.
replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.

StringBufferExample.java

```
1. import java.io.*;
2. import java.lang.*;
3. import java.util.*;
4. class StringBufferExample{
5. public static void main(String args[]){
6. StringBuffer sb=new StringBuffer(" Hello");
```

```
7. sb.append("Java");//now original string is changed
8. System.out.println("After appending the string " + sb);//prints Hello Java
9. System.out.println("Length of the Srting" + sb.length( ));
10. System.out.println("capacity of the Srting" + sb.capacity( ));
    sb.insert(0,"hai");//now original string is changed
    System.out.println("after insertion " + sb);
11. System.out.println("Reverse of the Srting" + sb.reverse( ));
    }
    }
```

Output:

```
After appending the string HelloJava
Length of the string 9
Capacity of the string 32
After insertion  haiHelloJava
Reverse of the String avaJolloeHiah
```

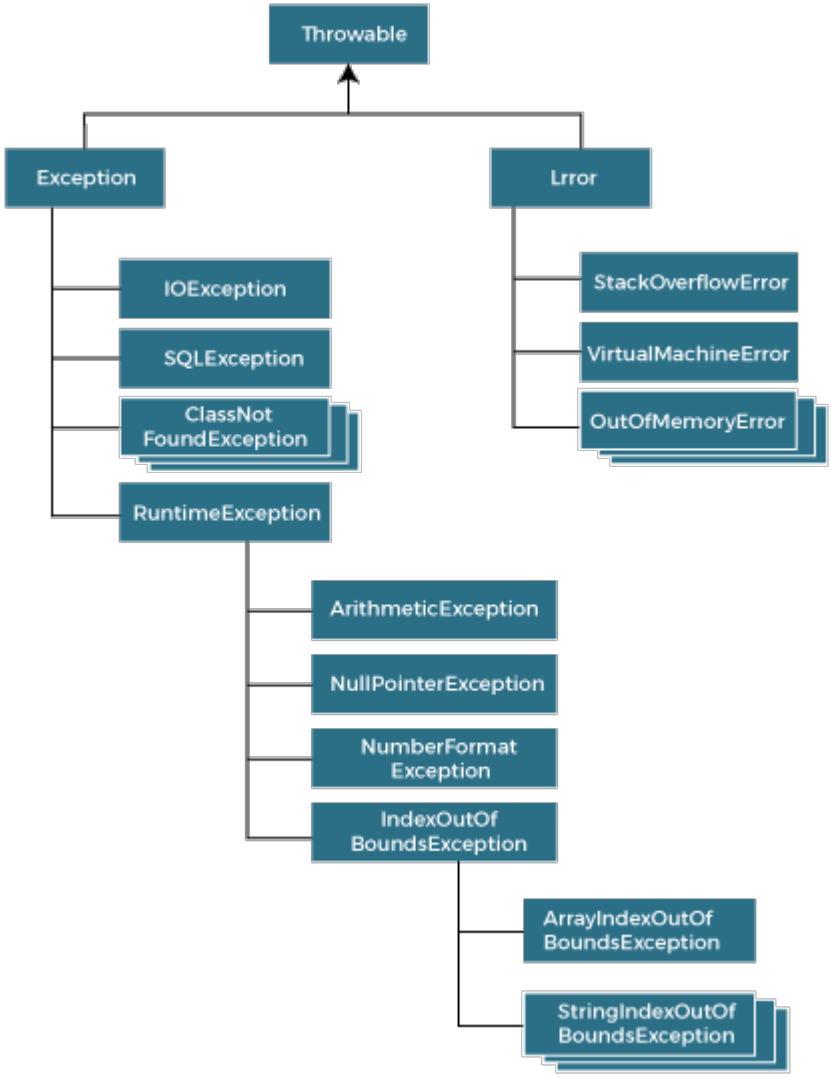
Exception Handling in Java

What is Exception Handling?

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

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:



Types of Java Exceptions

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

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

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.

Java try-catch block

Syntax of Java try-catch

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

Syntax of try-finally block

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

TryCatchExample2.java

```
public class TryCatchExample2 {

  public static void main(String[ ] args) {
    try
    {
    int data=50/0; //may throw exception
    }
      //handling the exception
    catch(ArithmeticException e)
    {
```

```
11.            System.out.println(e);
12.        }
13.        System.out.println("rest of the code");
14.    }
15.
16. }
```

Output:

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

Java Catch Multiple Exceptions

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.

MultipleCatchBlock2.java

```
1.  public class MultipleCatchBlock2 {
2.
3.      public static void main(String[] args) {
4.
5.          try{
6.              int a[]=new int[5];
7.
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException  e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
18.            catch(Exception e)
```

```
19.             {
20.              System.out.println("Parent Exception occurs");
21.             }
22.           System.out.println("rest of the code");
23.    }
24. }
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Java finally block

- **Java finally block** is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or notThe finally block follows the try-catch block.

TestFinallyBlock2.java

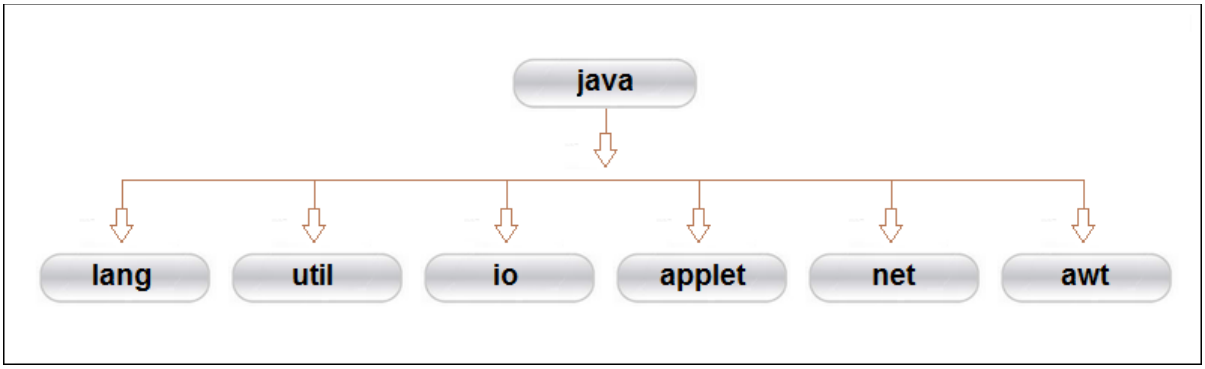
```
1.  public class TestFinallyBlock2{
2.      public static void main(String args[]){
3.
4.      try {
5.
6.       System.out.println("Inside try block");
7.
8.       //below code throws divide by zero exception
9.       int data=25/0;
10.      System.out.println(data);
11.     }
12.
13.     //handles the Arithmetic Exception / Divide by zero exception
14.     catch(ArithmeticException e){
15.      System.out.println("Exception handled");
16.      System.out.println(e);
17.     }
18.
```

```
19.     //executes regardless of exception occured or not
20.     finally {
21.       System.out.println("finally block is always executed");
22.     }
23.
24.     System.out.println("rest of the code...");
25.     }
26.   }
```

Java-API Packages

API Packages

Java **APl(Application Program Interface)** provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the the Java API. Following figure shows the system



packages that are frequently used in the programs.

Java System Packages and Their Classes

java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for creating and implementing applets.

java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

Creating and Accessing Packages:

Syntax:

```
1. package packagename;
2. public class classname
3. {
4. // Method definition;
5.
6.    }
7.
```

Simple example of package creation

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

```
8. //save as Simple.java
9. package mypack;
10. public class Simple{
11.  public static void main(String args[]){
12.    System.out.println("Welcome to package");
13.   }
14. }
```

1. How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
1. javac -d directory javafilename
```

For **example**

1. 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).

How to run java package program

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.

How to access package from another package?

There are two ways to access the package from outside the package.

1. import package.*;
2. import package.classname;

*1) Using packagename.**

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

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.*

```
1. //save by A.java
2. package pack;
3. public class A{
4.   public void msg()
5. {
```

```
6.  System.out.println("this is my package");
7.  }
8.  }
1.  //save by B.java
2.  import java.io.*;
3.  import java.util.*;
4.  import pack.*;
5.
6.  class B{
7.    public static void main(String args[]){
8.     A obj = new A();
9.     obj.msg();
10.   }
11. }
Output:Hello
```

2) 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

```
1.  //save by A.java
2.  package pack;
3.  public class A {
4.   public void msg(){System.out.println("Hello");
5.  }
6.  }
```

```
//save by B.java
import java.io.*;
import java.util.*;
import pack.A;

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

```
    obj.msg();
1.  }
2. }
```

```
Output:Hello
```

Assertions in Java

- An assertion allows testing the correctness of any assumptions that have been made in the program.
- An assertion is achieved using the **assert** statement in Java. While executing assertion, it is believed to be true.
- If it fails, JVM throws an error named **AssertionError.**
- It is mainly used for testing purposes during development.

Enabling Assertions

By default, assertions are disabled. We need to run the code as given. The syntax for enabling assertion statement in Java source code is:

java –ea Test

Or

java –enableassertions Test

Here, Test is the file name.

Disabling Assertions

The syntax for disabling assertions in java is:

java –da Test

Or

java –disableassertions Test

Here, Test is the file name.

Why use Assertions

Wherever a programmer wants to see if his/her assumptions are

wrong or not.

- To make sure that an unreachable-looking code is actually unreachable.
- To make sure that assumptions written in comments are right.

```
\\Java program to demonstrate syntax of assertion

import java.util.Scanner;
class Test {
    public static void main(String args[])
    {
        int age = 15;
        assert age< 18 : " Unable to vote";
        System.out.println("The age is " + age);
    }
}
```

Output
The age is 15

After enabling assertions:
Output:
Exception in thread "main" java.lang.AssertionError: Unable to vote