Object Oriented Programming(OOP)

Class

A class is a group of objects which have common properties. It is a **template or blueprint** from which objects are created. It is a logical entity. Class does not occupy memory. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Interface

Class Declaration

```
class <class name>
{
        // Code
}
```

Object

- An object is an instance of a class.
- An object is a real-world entity.
- The object is an entity which has state and behavior.

Object Creation:

<Class_name> object = new <Class_name>();

Encapsulation

Encapsulation refers to the **bundling of fields and methods inside a single class.**

It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve **data hiding.**

In Java, encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.

```
class Employee
{
   private int empId;
   private String empName;
   private int empSalary;

   public int getEmpId(){
      return empId;
   }

   public void setEmpId(int Id)
   {
      this.empId = Id;
   }

   public String getEmpName()
   {
      return empName;
   }
   public void setEmpName(String name){
      this.empName = name;
   }

    public int getEmpSalary()
   {
      return empSalary;
   }
   public void setEmpSalary(int salary){
      this.empSalary = salary;
    }


}


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

        Employee obj = new Employee();

        obj.setEmpId(1000);
        obj.setEmpName("Arun");
        obj.setEmpSalary(30000);

        System.out.println("name:" + obj.getEmpId());
        System.out.println("Salary:" + obj.getEmpSalary());

    }
}
```

Inheritance

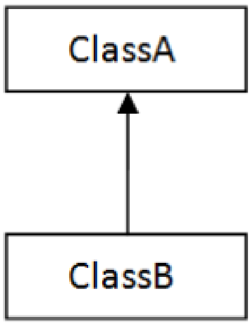
Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.When you inherit from an existing class, you can reuse methods and fields of the parent class.

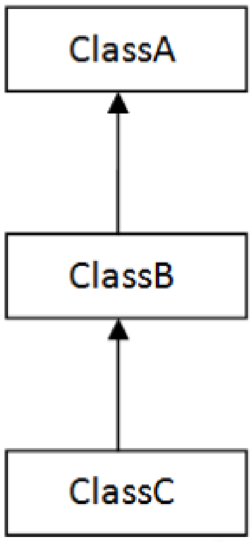
Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

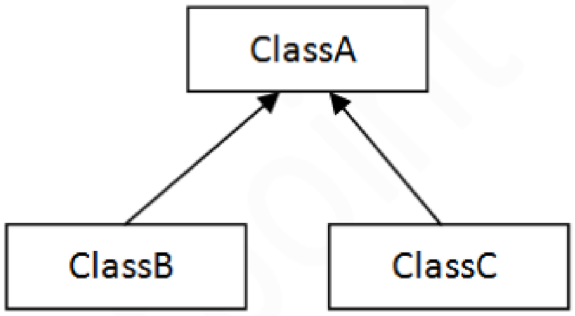
Multiple Inheritance
Hybrid Inheritance



1) Single

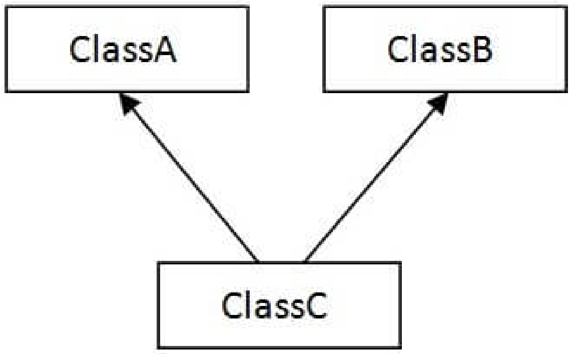


2) Multilevel

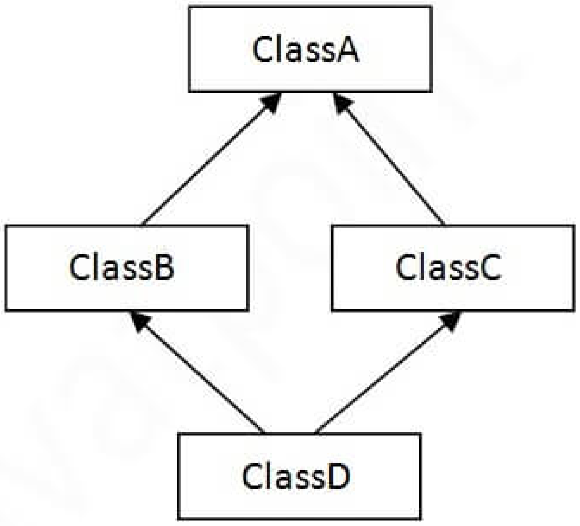


3) Hierarchical

NOTE: Object creation for least child class



4) Multiple



5) Hybrid

Parent class
Base class - - > Base class
Super class

Derived class
Child class - - > Child class
Sub class

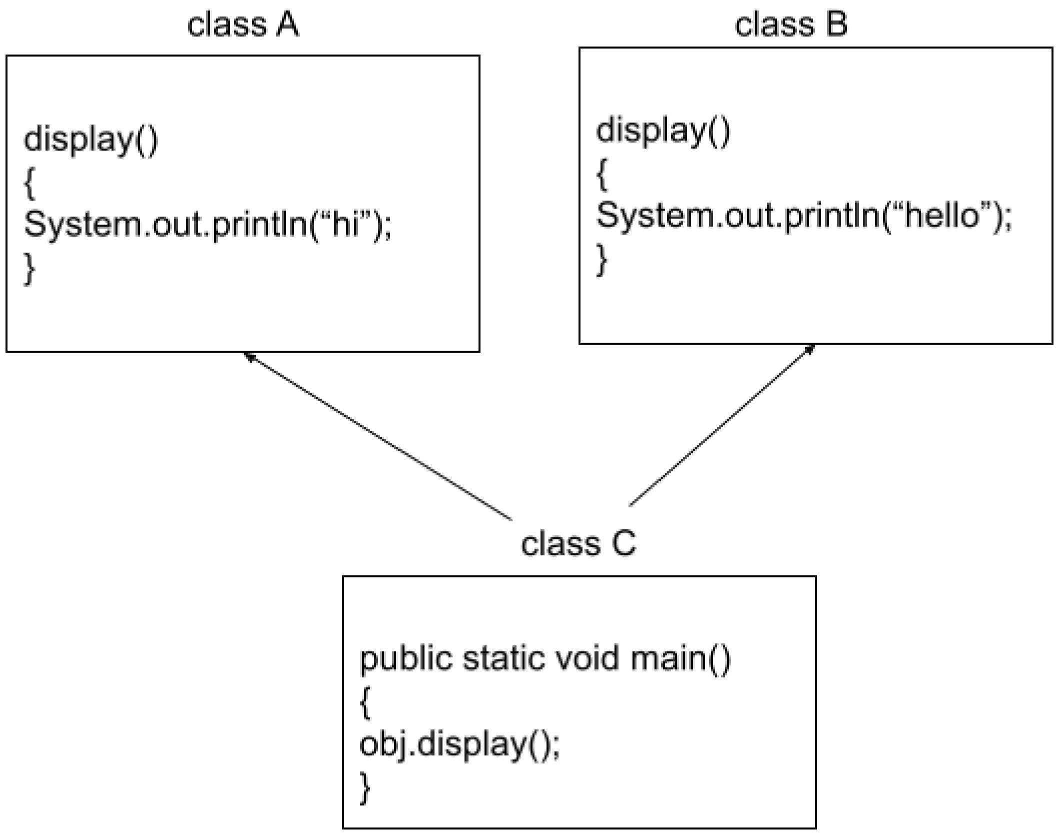
```
import java.util.*;
class StudentDetails
{
int stuId;
String name;
Scanner scan=new Scanner(System.in);

void getDetails(){
System.out.println("------------------- Student Mark Calculation System -------------------");
System.out.println("Type Student ID");
stuId=scan.nextInt();

System.out.println("Type Student Name");
name=scan.nextLine();
}

}

class StudentMarks extends StudentDetails{
int m1, m2, m3, total;

void calculateResult(){
System.out.println("Type Mark1");
m1=scan.nextInt();
System.out.println("Type Mark2");
m2=scan.nextInt();
System.out.println("Type Mark3");
m3=scan.nextInt();
total=m1+m2+m3;
System.out.println("********* MarkReport *********");
System.out.println("Id:"+stuId+"\nName:"+name+"\nTotal:"+total+"\n****************************");
}

}

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

StudentMarks obj = new StudentMarks();
obj.getDetails();
obj.calculateResult();

}}
```

Why multiple inheritance is not supported in Java?

The reason behind this is to prevent ambiguity.

Consider a case where class C extends class A and Class B and both class A and B have the same method display().

Now the Java compiler cannot decide which display method it should inherit. To prevent such a situation, multiple inheritance is not allowed in java.



```
display()
{
System.out.println("hi");
}
```

```
display()
{
System.out.println("hello");
}
```

```
public static void main()
{
obj.display();
}
```

Polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*.

Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java:

- Compile-time polymorphism
- Runtime polymorphism.

Compile Time polymorphism is implemented through **Method overloading**.
Run time polymorphism is implemented through **Method overriding.**

Compile-time Polymorphism

Method Overloading occurs when a class has many methods with the same name but different parameters. Two or more methods may have the same name if they have other **numbers of parameters, different data types, or different numbers of parameters and different data types.**

```
public class MethodOverloading {

        void show(int num1)
        {
                System.out.println("number 1 : " + num1);
        }

        void show(int num1, int num2)
        {
                System.out.println("number 1 : " + num1
                                                + " number 2 : " + num2);
        }

        public static void main(String[] args)
        {
                MethodOverloading obj = new MethodOverloading();
                obj.show(3);
                obj.show(4, 5);
        }
}
```

Runtime Polymorphism

In this process, an **overridden method is called through the reference variable of a superclass**. The determination of the method to be called is based on the object being referred to by the reference variable.

```
class Bike{
  void run(){
     System.out.println("running");
  }
}
class Splendor extends Bike{
  void run(){
      System.out.println("running safely with 60km");
}


  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Abstraction

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

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class

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.

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.

```
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();
}
}
```

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.

```
interface <interface_name>{
        // declare constant fields
        // declare methods that abstract
}
class class_name implements interface_name{


}
```

```
interface Drawable{
void draw();
}

class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}

class InterfaceExample{
public static void main(String args[]){
Drawable d=new Circle();
d.draw();
}}
```

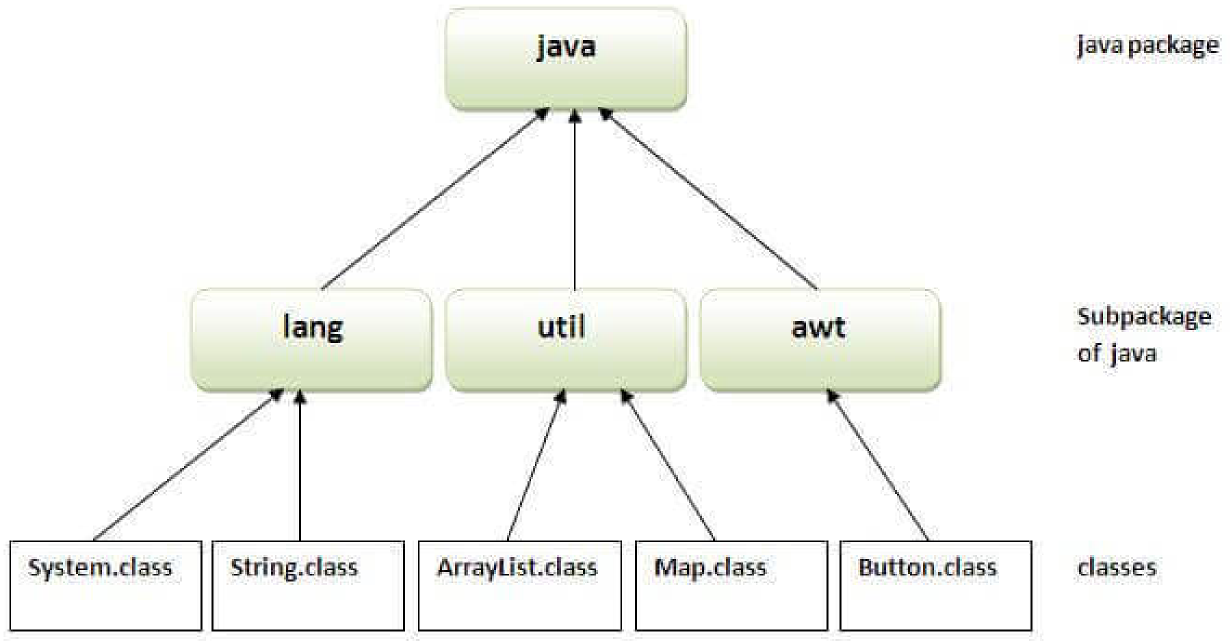
Packages

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.

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.



compile java package

```
javac -d directory javafilename
```

1. We can include interface into Package
2. We can include inheritance into Package

```
package calculator;
public class Arith{

public int addition(int a, int b){

return a+b;

}

public int subtraction(int a, int b){

return a-b;

}

public int multiplication(int a, int b){

return a*b;
}


public int division(int a, int b){

return a/b;

}

public static void display()

{

System.out.println(“this is static method”);

}

}


—-----------------------------------------------------------------------

import calculator.Arith;

class PackageAccess{

public static void main(String[] args){

int a=10, b=20;

Arith obj = new Arith();

System.out.println(obj.addition(a,b));

System.out.println(obj.subtraction(a,b));

System.out.println(obj.multiplication(a,b));

System.out.println(obj.division(a,b));

}

}
```

Constructor

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created.

It is a special type of method which is used to initialize the object.

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

There are two types of constructors in Java:

- Default (no-arg) constructor
- Parameterized constructor.

Default constructor

```
class Bike{
Bike(){
   System.out.println("Bike is created");
}
public static void main(String args[]){
Bike b=new Bike();
}
}
```

Parameterized constructor

```
class Student{
   int id;
   String name;
   Student(int i,String n){
   this.id = i;
   this.name = n;
   }
   void display(){
```

```
   System.out.println(id+" "+name);
}

   public static void main(String args[]){

   Student s1 = new Student(101,"Arjun");
   Student s2 = new Student(102,"Bala");

   s1.display();
   s2.display();
  }
}
```

Aggregation(HAS-A relationship)

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

```
class Employee{

int id;

String name;

Address address;//Address is a class

...

}
```

Address.java

```
public class Address {

String city,state,country;

public Address(String city, String state, String country) {

    this.city = city;

    this.state = state;

    this.country = country;

}

}
```

Emp.java

```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
    this.id = id;
    this.name = name;
    this.address=address;
}

void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}

public static void main(String[] args) {
Address address1=new Address("abc","TN","india");
Address address2=new Address("xyz","Ban","india");

Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);

e.display();
e2.display();

}
}
```

Array

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location.

It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

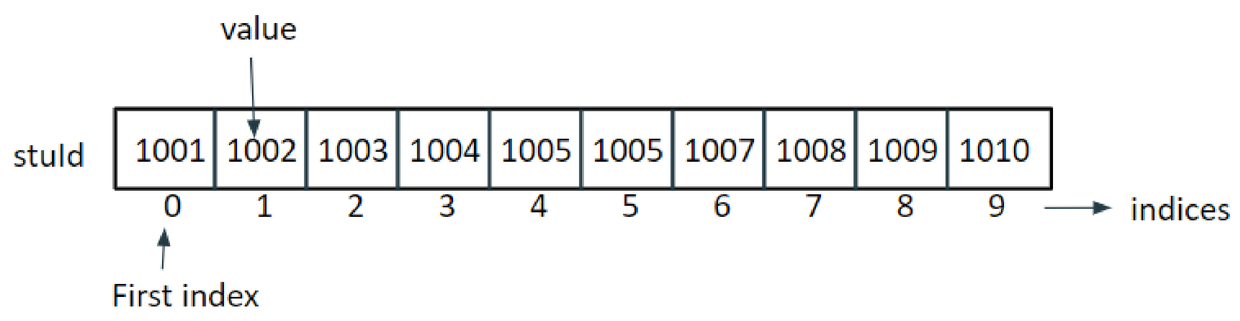
There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array

Array Declaration [ ]

1. dataType[] stuId;
2. dataType []stuId;
3. dataType stuId[];
4. stuId = new datatype[size];

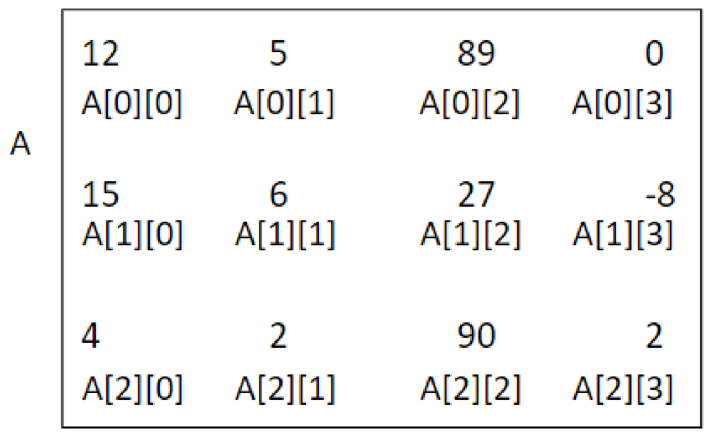


```
int stuId[] =  new int[10];

stuId[0] = 1001;

stuId[1] = 1002;

stuId[2] = 1003;

stuId[3] = 1004;

…..

stuId[9] = 1010;
```

Two Dimensional Array (2D array)

Array Declaration

1. `dataType[][]  variableName;`
2. `dataType   [][]variableName;`
3. `dataType  variableName[][];`
4. `dataType  []variableName[];`
5. `int[][] arr=new int[3][3];`


A

12 A[0][0]	5 A[0][1]	89 A[0][2]	0 A[0][3]
15 A[1][0]	6 A[1][1]	27 A[1][2]	-8 A[1][3]
4 A[2][0]	2 A[2][1]	90 A[2][2]	2 A[2][3]

```
int[][] A = new int[3][4];
```

```
import java.util.*;

class TwoDimensionalArray{

 public static void main(String[] args){

Scanner scan=new Scanner(System.in);

int[][] A=new int[3][3];


  int i,j;

  for(i=0;i<3;i++){

           for( j=0; j<3; j++){

           A[i][j] = scan.nextInt();

           }

     }

System.out.println("Matrix A is:");

    for(i=0;i<3;i++){

          for(j=0;j<3;j++){

               System.out.print(A[i][j]+" ");
```

```
        }

          System.out.print("\n");
        }

   }
}
```

Three Dimensional Array (3D array)

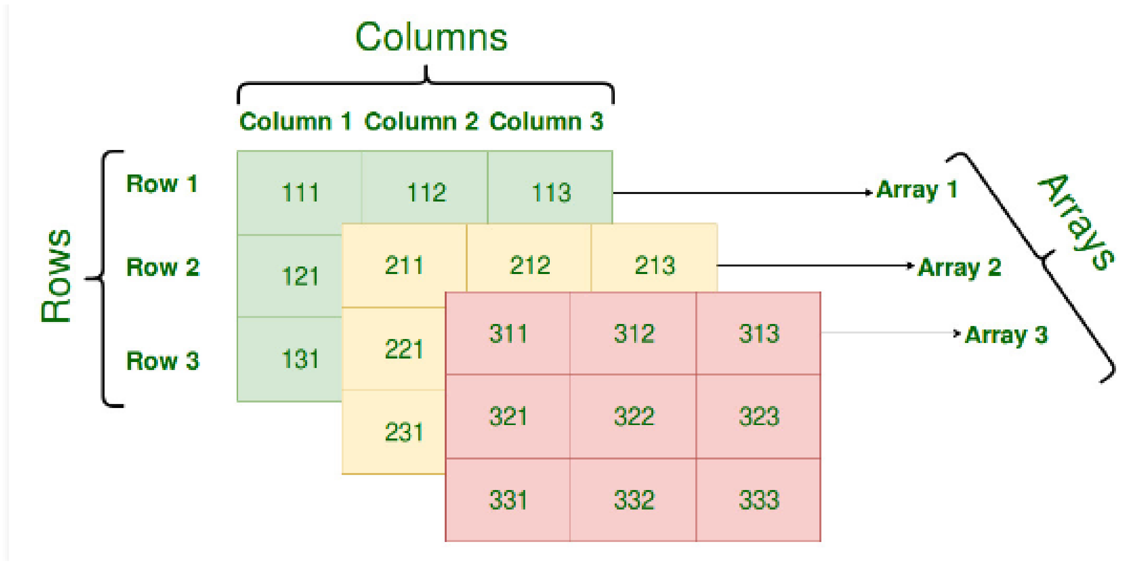
Elements in three-dimensional arrays are commonly referred by **x[i][j][k]** where ‘i’ is the array number, ‘j’ is the row number and ‘k’ is the column number.

Syntax:

```
x[array_index][row_index][column_index]
```

For example:

```
int[][][] x = new int[3][3][3];
```



```
int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };
```

```
class ThreeDimensionalArray {
    public static void main(String[] args)
    {
        int[][][] arr = {
                    { { 1, 2 }, { 3, 4 } },
                         { { 5, 6 }, { 7, 8 } }
                   };

        for (int i = 0; i < 2; i++) {

            for (int j = 0; j < 2; j++) {

                for (int k = 0; k < 2; k++) {

                    System.out.print(arr[i][j][k] + " ");
                }

                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Jagged Array

```
class Main {
          public static void main(String[] args)
          {


                    int arr[][] = new int[2][];

                    // First row has 3 columns
                    arr[0] = new int[3];

                    // Second row has 2 columns
                    arr[1] = new int[2];
```

```
                int count = 0;
                for (int i = 0; i < arr.length; i++)
                        for (int j = 0; j < arr[i].length; j++)
                                arr[i][j] = count++;

                // Displaying the values of 2D Jagged array
                System.out.println("Contents of 2D Jagged Array");
                for (int i = 0; i < arr.length; i++) {
                        for (int j = 0; j < arr[i].length; j++)
                                System.out.print(arr[i][j] + " ");
                        System.out.println();
                }
        }
}
```

String

In Java, string is basically an object that **represents sequence of char values.**

The Java String is **immutable** which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

There are two ways to create String object:

1. By string literal
2. By new keyword

```
String s="welcome";   // string literal

String s=new String("Welcome")  // create using new keyword
```

```
public class StringExample{

public static void main(String args[]){

String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);  //converting char array to string
String s3=new String("example");  //creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

char charAt(int index)	It returns char value for the particular index
int length()	It returns string length
String substring(int beginIndex)	It returns substring for given begin index.
String replace(char old, char new)	It replaces all occurrences of the specified char value.
int indexOf(char ch)	It returns the specified char value index.
String toLowerCase()	It returns a string in lowercase.
String toUpperCase()	It returns a string in uppercase.
String trim()	It removes beginning and ending spaces of this string.
boolean equals(Object another)	It checks the equality of string with the given object.

```
class StringExample{

    public static void main(String[] args) {



        String text = "    HI, This is Sample Text    ";

        int len=text.length();

        //System.out.println(len);

      // System.out.println(text.charAt(4));

      // System.out.println(text.length());

       // System.out.println(text.substring(6, 10));

        //System.out.println();

       // String text2 = text.replace('I', 'i');

       // text = text.replace('I', 'i');

//         System.out.println(text);

       // System.out.println(text.trim());

String text2="java";

System.out.println(text.equals(text2));



    }

}
```

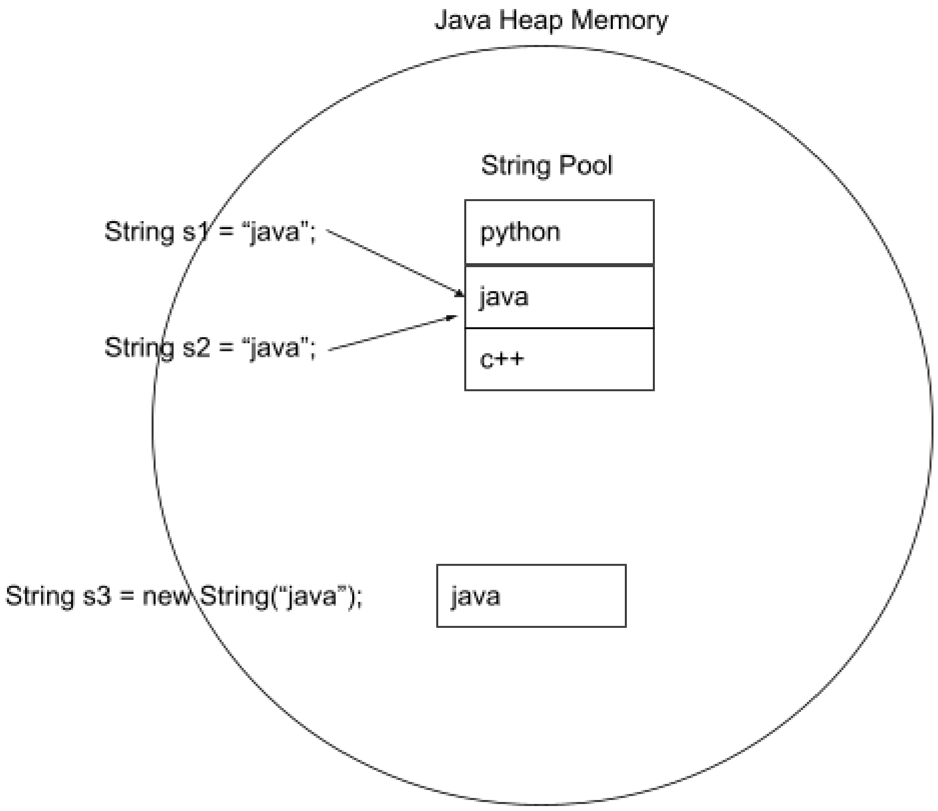
equals() vs ==

The Java String class **equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

```
String s1=”java”;
String s2=”java”;
System.out.println(s1.equals(s2)); // true
```

In simple words, **==** checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

```
String s1=”java”;
String s2=”java”;
String s3 = new String(“java”);
System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
```



StringBuffer

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.

Text = “credo”

text.append(“systems”); //Credo systems

text.insert(2,”systems”); // Crsystemsedo

- append(String s)
- insert(int offset, String s)
- replace(int startIndex, int endIndex, String str)
- delete(int startIndex, int endIndex)
- reverse()
- substring(int beginIndex)
- length()

StringBuilder

Java StringBuilder class is used to create **mutable** (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

- append(String s)
- insert(int offset, String s)
- replace(int startIndex, int endIndex, String str)
- delete(int startIndex, int endIndex)
- reverse()
- substring(int beginIndex)
- length()

Exception Handling

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

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

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**.

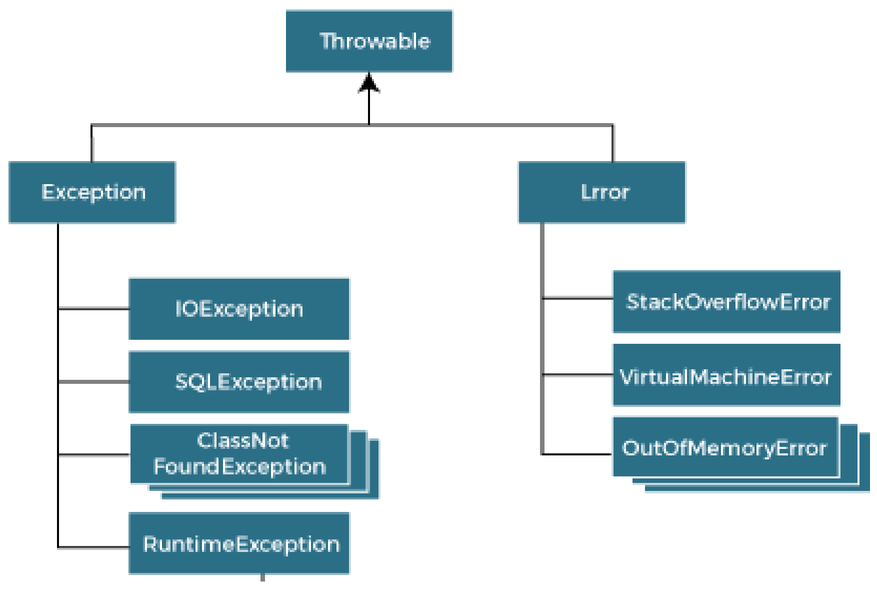
Unchecked Exception

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**.

Error

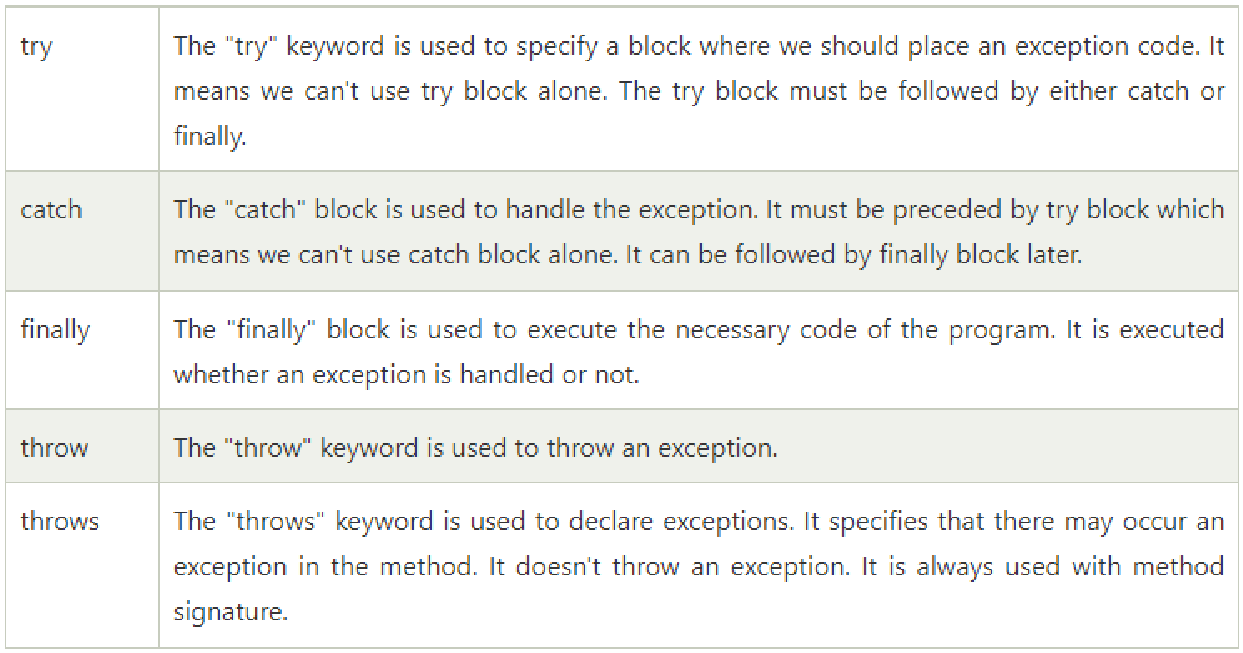
Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

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:



```
import java.util.*;

class ExceptionHandlingExample {

   public static void main(String[] args) {

      int n, result=0;

      Scanner scan=new Scanner(System.in);

      System.out.println("Type n value");

      n=scan.nextInt();

      try{

      result = 100/n;

      }

      catch(Exception e){

          System.out.println("Can't divide by Zero. So plz enter valid number");

          n=scan.nextInt();

          result = 100/n;
```

```
            System.out.println(result);
        }
        finally{
            System.out.println(result);
        }

    }
}
```

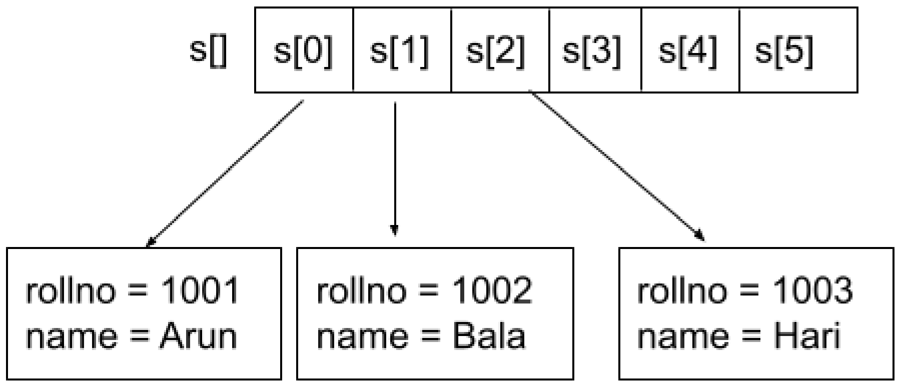


try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use 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.

```
 import java.util.*;
public class ExceptionHandling {
    public static void main(String[] args)
    {
        int age;
        Scanner scan=new Scanner(System.in);
        System.out.println(“Type your age”);
        age=scan.nextInt();

        try {

            if(age<18){

            throw new ArithmeticException("You are not eligible");

            }

        }

        catch (ArithmeticException e) {

            e.printStackTrace();

           // System.out.println("Type age");

            //age=scan.nextInt();

        }

    }

}
```

Array of objects

```
class Student

{

int rollno;

String name;

}
```



```
class ArrayOfObjectsExample {

        public static void main(String args[])
        {

                Student[] arr;

                arr = new Student[2];
                arr[0] = new Student(1701289270, "Satyabrata");

                arr[1] = new Student(1701289219, "Omm Prasad");

                System.out.println("Student data in student arr 0: ");
                arr[0].display();

                System.out.println("Student data in student arr 1: ");
                arr[1].display();
        }
}

class Student {

        public int id;
        public String name;
        Student(int id, String name)
        {
                this.id = id;
                this.name = name;
        }
        public void display()
        {
                System.out.println("Student id is: " + id + " "
                                                + "and Student name is: "
                                                + name);
                System.out.println();
        }
}
```

Java Updated Features

Varargs

The varrags allows the method **to accept zero or multiple arguments**. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

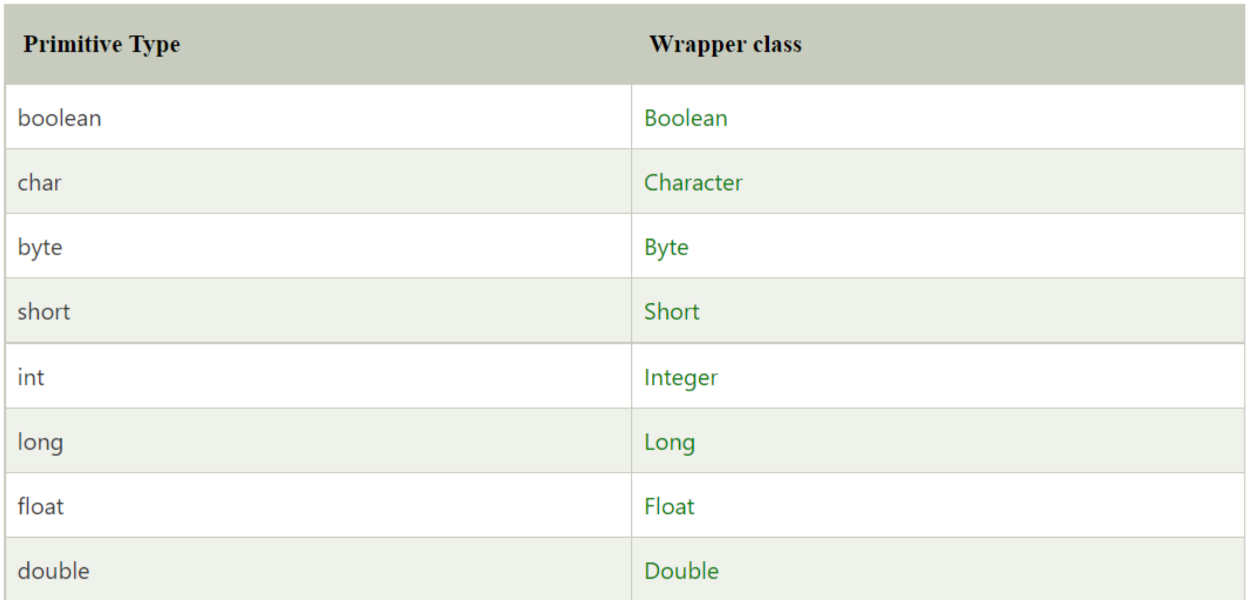
Syntax

Three dots after the data type. Syntax is as follows:

```
return_type method_name(data_type... variableName)
{
}
```

Wrapper Class

The wrapper class in Java provides the mechanism *to convert primitive into object and object into primitive*.



Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

The automatic **conversion of primitive data type into its corresponding wrapper class** is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

AutoUnboxing

The automatic **conversion of wrapper type into its corresponding primitive type** is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

```
public class WrapperClassExample{
public static void main(String args[]){

        int a = 20; Integer i = Integer.valueOf(a);// converting int into Integer  explicitly
        Integer j = a;// autoboxing, now compiler will write Integer.valueOf(a) internally

        System.out.println(a + " " + i + " " + j);

// ------------- unboxing----------
       Integer a = new Integer(3);
       int i = a.intValue();// converting Integer to int explicitly
       int j = a;// unboxing, now compiler will write a.intValue() internally

       System.out.println(a + " " + i + " " + j);



}}
```

Enum

The Enum in Java is a data type which contains a **fixed set of constants**.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java.

```
class EnumExample{

	public enum Season {
		WINTER, SPRING, SUMMER, FALL
	}

	public static void main(String[] args) {

		System.out.println(Season.values());

		for (Season s : Season.values()) {
			System.out.println(s);
		}

		System.out.println("Value of WINTER is: " + Season.valueOf("WINTER"));
		System.out.println("Index of WINTER is: " +
Season.valueOf("WINTER").ordinal());
		System.out.println("Index of SUMMER is:
"+Season.valueOf("SUMMER").ordinal());

	}}
```

Annotation

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

Built-In Java Annotations used in Java code

- @Override
- @SuppressWarnings
- @Deprecated

Custom annotations

Java Custom annotations or Java User-defined annotations are easy to create and use. The @*interface* element is used to declare an annotation.

There are few points that should be remembered by the programmer.

1. Method should not have any throws clauses
2. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3. Method should not have any parameter.
4. We should attach @ just before interface keyword to define annotation.
5. It may assign a default value to the method.

Types

There are three types of annotations.

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

Marker Annotation

An annotation that has no method, is called marker annotation.The @Override and @Deprecated are marker annotations.

```
@interface MyAnnotation

{

}
```

Single value Annotation

An annotation that has one method, is called single-value annotation.

```
@interface MyAnnotation{

int value();

}
```

Multi value Annotation

An annotation that has more than one method, is called Multi-Value annotation.

```
@interface MyAnnotation{

int value1();

String value2();

String value3();

}
```

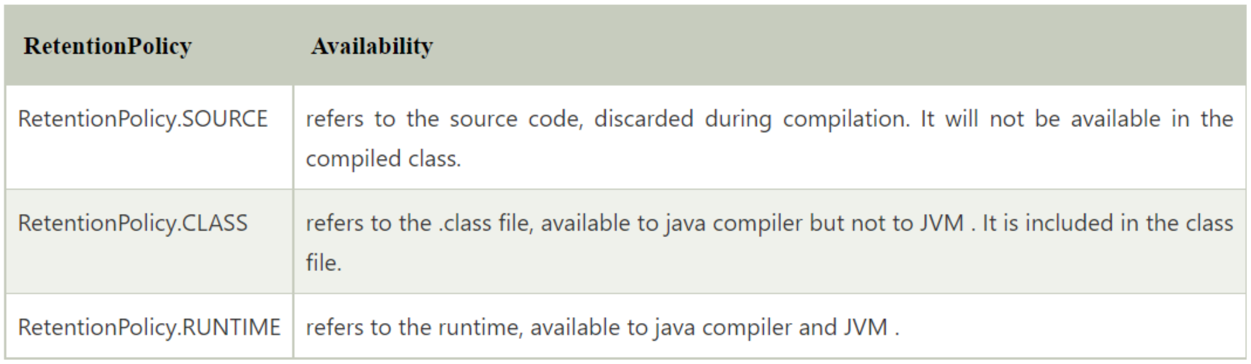
Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as,
TYPE,
METHOD,
FIELD
LOCAL_VARIABLE
ANNOTATION_TYPE
PARAMETER

@Retention annotation is used to specify to what level annotation will be available.



RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

@Documented

The @Documented Marks the annotation for inclusion in the documentation.

Assertion

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing the assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purposes.

Syntax

```
assert expression;

        or

assert expression1 : expression2;
```

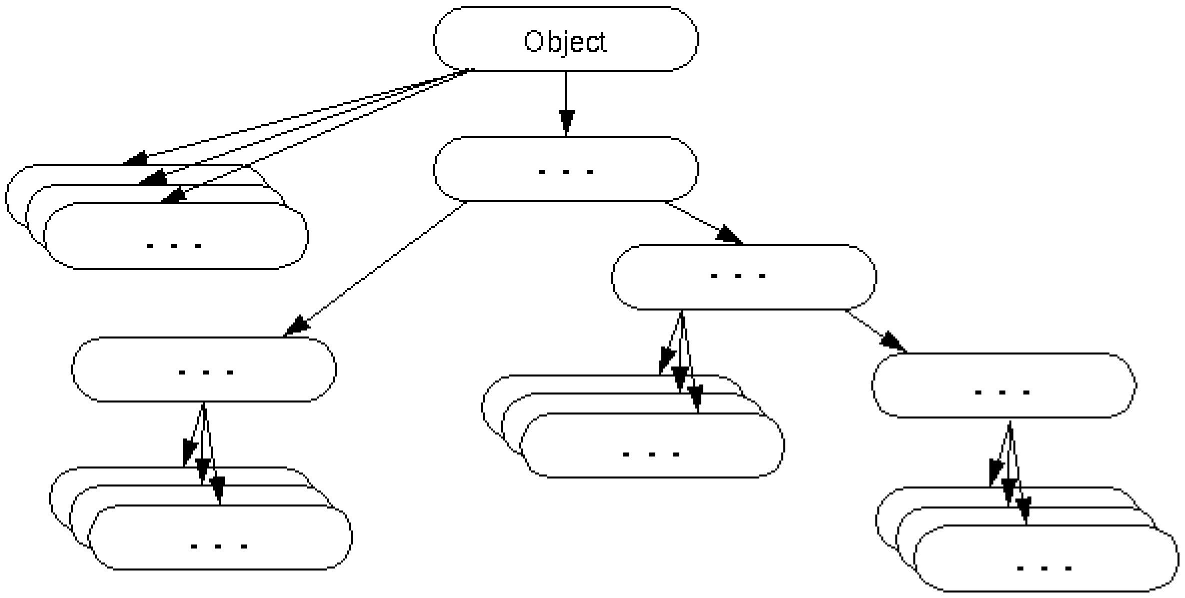
If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, -ea or -enableassertions switch of java must be used.
Compile it by: **javac AssertionExample.java**
Run it by: **java -ea AssertionExample**

```
import java.util.Scanner;

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

  Scanner scanner = new Scanner( System.in );
  System.out.print("Enter ur age ");

  int value = scanner.nextInt();
  assert value>=18:" Not valid";

  System.out.println("value is "+value);
 }
}
```

OOPS Miscellaneous

Object class

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a class does not extend any other class then it is a direct child class of Object and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes.



The Object class provides multiple methods which are as follows:

- tostring() method
- hashCode() method
- equals(Object obj) method
- finalize() method
- getClass() method
- clone() method
- wait(), notify() notifyAll() methods

Object cloning

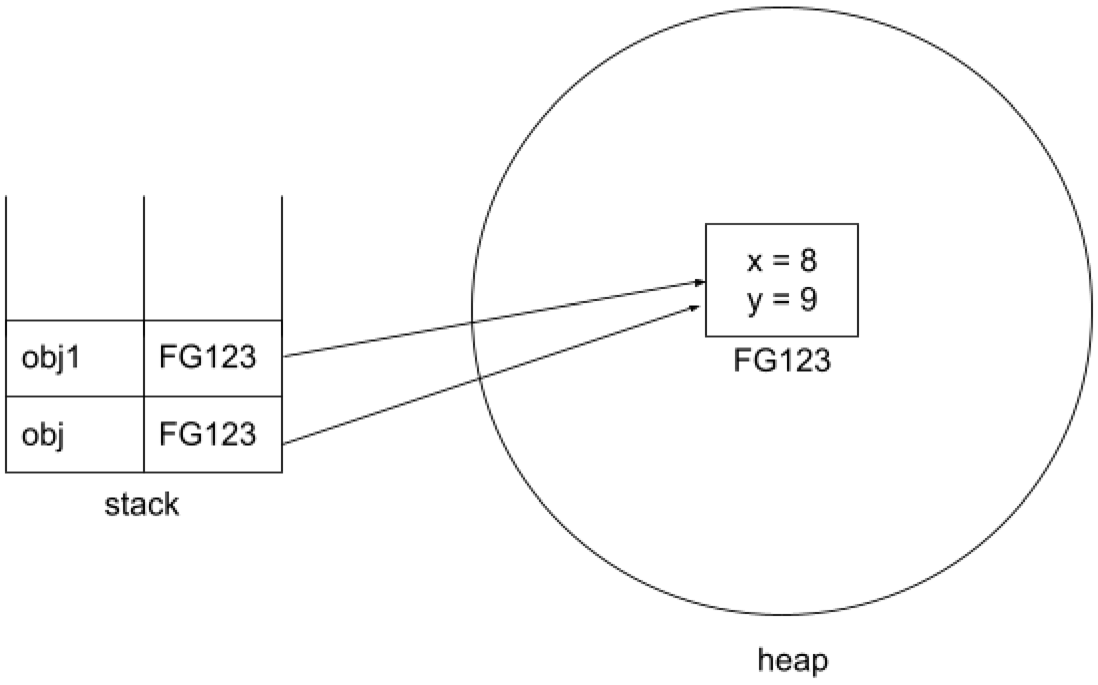
There are three types of object copying.

1. Shallow copy
2. Deep copy
3. Clone

Shallow copy

A shallow copy of an object is a new object whose instance variables are identical to the old object. For example, a shallow copy of a Set has the same members as the old Set and shares objects with the old Set through pointers. Shallow copies are sometimes said to use reference semantics.

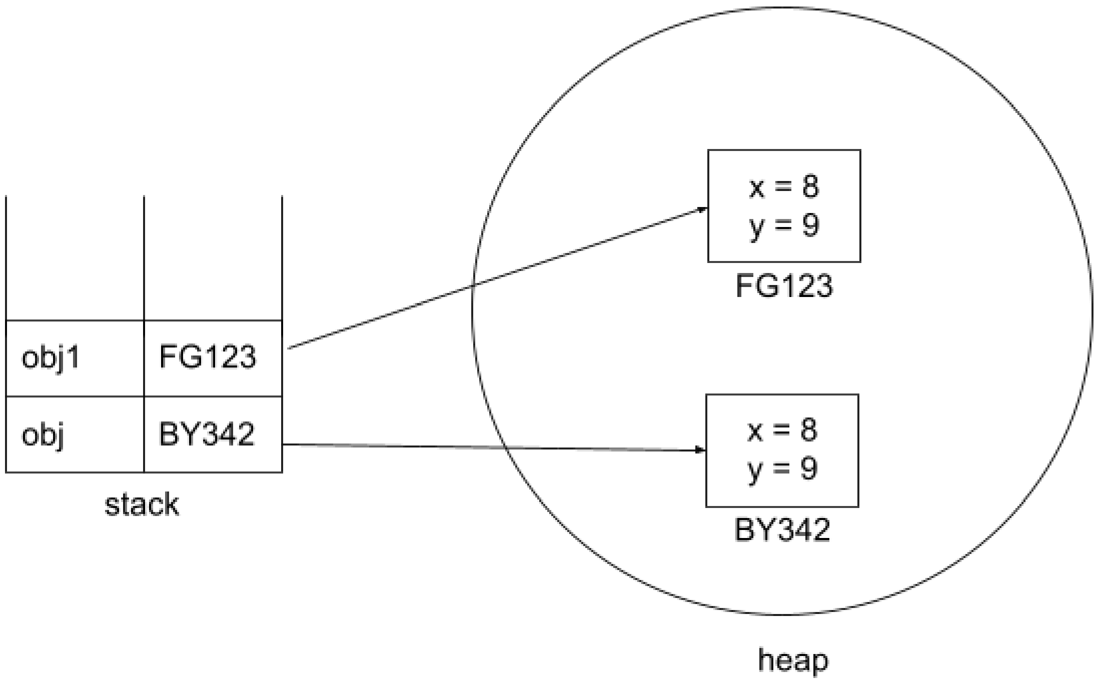
```
class Abc
{
int x;
int y;
}
Abc obj = new Abc();
obj.x = 8;
obj.y = 9;

Abc obj1 = obj;    // shallow copy
```



Shallow copy

Deep copy

When we do a copy of some entity to create two or more than two entities such that changes in one entity are not reflected in the other entities, then we can say we have done a deep copy. In the deep copy, a new memory allocation happens for the other entities, and reference is not copied to the other entities. Each entity has its own independent reference.



Deep copy