

# JAVA OOP CONCEPTS

## Object and Classes

Since Java is an object oriented language, complete java language is build on classes and object. Java is also known as a strong **Object oriented programming language**(oops).

OOPS is a programming approach which provides solution to problems with the help of algorithms based on real world. It uses real world approach to solve a problem. So object oriented technique offers better and easy way to write program then procedural programming model such as C, ALGOL, PASCAL.

---

### Main Features of OOPS

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

As an object oriented language Java supports all the features given above. We will discuss all these features in detail later.

---

## Class

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. Once defined this new type can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class.

A class is declared using **class** keyword. A class contain both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**.

---

## Rules for Java Class

- A class can have only public or default(no modifier) access specifier.
  - It can be either abstract, final or concrete (normal class).
  - It must have the class keyword, and class must be followed by a legal identifier.
  - It may optionally extend one parent class. By default, it will extend java.lang.Object.
  - It may optionally implement any number of comma-separated interfaces.
  - The class's variables and methods are declared within a set of curly braces `{}`.
  - Each **.java** source file may contain only one public class. A source file may contain any number of default visible classes.
  - Finally, the source file name must match the public class name and it must have a .java suffix.
- 

## A simple class example

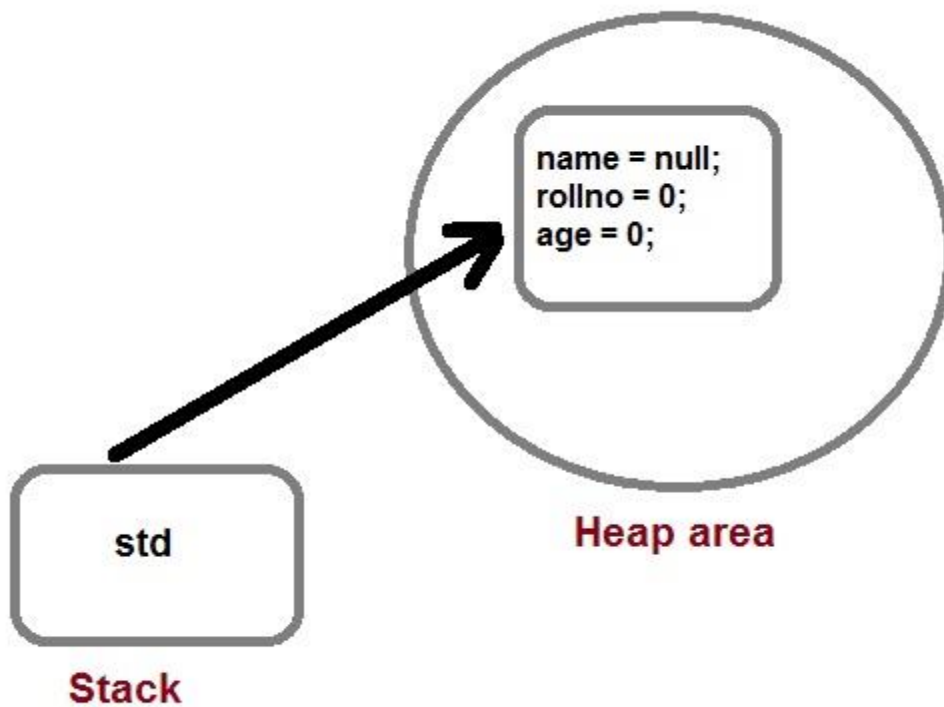
Suppose, Student is a **class** and student's name, roll number, age will be its property. Lets see this in Java syntax

```
class Student.  
{  
    String name;  
    int rollno;  
    int age;  
}
```

When a reference is made to a particular student with its property then it becomes an **object**, physical existence of Student class.

```
Student std=new Student();
```

After the above statement **std** is instance/object of Student class. Here the **new** keyword creates an actual physical copy of the object and assign it to the **std** variable. It will have physical existence and get memory in heap area. The **new** operator dynamically allocates memory for an object



---

### Q. How a class is initialized in java?

A Class is initialized in Java when an instance of class is created using either **new** operator or using reflection using `Class.forName()`. A class is also said to be initialized when a static method of **Class** is invoked or a static field of **Class** is assigned.

---

### Q. How would you make a copy of an entire Java object with its state?

Make that class implement **Cloneable** interface and call **clone()** method on its object. **clone()** method is defined in **Object** class which is parent of all java class by default.

## Methods in Java

Method describe behavior of an object. A method is a collection of statements that are group together to perform an operation.

**Syntax :**

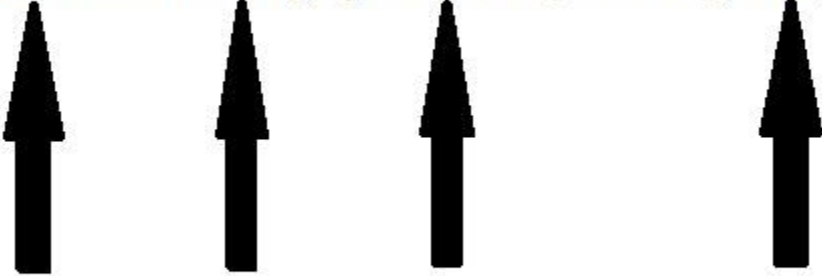
```
return-type methodName(parameter-list)
{
    //body of method
}
```

---

**Example of a Method**

```
public String getName(String st)
{
    String name="StudyTonight";
    name=name+st;
    return name;
}
```

`public String getName(String st)`



`modifier   return-type   method-name   parameter`

**Modifier :** Modifier are access type of method. We will discuss it in detail later.

**Return Type :** A method may return value. Data type of value return by a method is declare in method heading.

**Method name :** Actual name of the method.

**Parameter :** Value passed to a method.

**Method body** : collection of statement that defines what method does.

---

## Parameter Vs. Argument

While talking about method, it is important to know the difference between two terms **parameter** and **argument**.

**Parameter** is variable defined by a method that receives value when the method is called.

Parameter are always local to the method they dont have scope outside the method.

While **argument** is a value that is passed to a method when it is called.

```
public void sum( int x, int y )
{
    System.out.println(x+y);
}
public static void main( String[ ] args )
{
    Test b=new Test( );
    b.sum( 10, 20 );
}
```

The diagram shows the word "parameter" in red with a line pointing to the parameter "int y" in the sum method signature. It also shows the word "argument" in red with a line pointing to the value "20" in the method call "b.sum( 10, 20 );".

---

## call-by-value and call-by-reference

There are two ways to pass an argument to a method

1. **call-by-value** : In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.

2. **call-by-reference** : In this reference of an argument is pass to a method. Any changes made inside the method will affect the argument value.

**NOTE** : In Java, when you pass a primitive type to a method it is passed by value whereas when you pass an object of any type to a method it is passed as reference.

---

### Example of call-by-value

```
public class Test
{
    public void callByValue(int x)
    {
        x=100;
    }
    public static void main(String[] args)
    {
        int x=50;
        Test t = new Test();
        t.callByValue(x);    //function call
        System.out.println(x);
    }
}
```

Output : 50

---

### Example of call-by-reference

```
public class Test
{
```

```

int x=10;
int y=20;

public void callByReference(Test t)
{
    t.x=100;
    t.y=50;
}

public static void main(String[] args)
{

    Test ts = new Test();
    System.out.println("Before "+ts.x+" "+ts.y);
    ts.callByReference(ts);
    System.out.println("After "+ts.x+" "+ts.y);
}
}

```

**Output :**

Before 10 20

After 100 50

---

## Method overloading

If two or more method in a class have same name but different parameters, it is known as method overloading.

Method overloading is one of the ways through which java supports polymorphism. Method overloading can be done by changing number of arguments or by changing the data type of arguments. If two or more method have same name and same parameter list **but differs in return type are not** said to be overloaded method

---

## Different ways of Method overloading

There are two different ways of method overloading

### Method overloading by changing data type of Arguments

*Example :*

```
class Calculate
{
    void sum (int a, int b)
    {
        System.out.println("sum is" +(a+b)) ;
    }
    void sum (float a, float b)
    {
        System.out.println("sum is" +(a+b));
    }
    Public static void main (String[] args)
    {
        Calculate cal = new Calculate();
        cal.sum (8,5);      //sum(int a, int b) is method is called.
        cal.sum (4.6, 3.8); //sum(float a, float b) is called.
    }
}
```

**Output:**

Sum is 13

Sum is 8.4

You can see that sum() method is overloaded two times. The first takes two integer arguments, the second takes two float arguments.

---



## Method overloading by changing no. of argument.

*Example :*

```
class Area
{
    void find(int l, int b)
    {
        System.out.println("Area is"+(l*b)) ;
    }
    void find(int l, int b,int h)
    {
        System.out.println("Area is"+(l*b*h));
    }
    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(8,5);    //find(int l, int b) is method is called.
        ar.find(4,6,2);  //find(int l, int b,int h) is called.
    }
}
```

**Output:**

Area is 40

Area is 48

In this example the find() method is overloaded twice. The first takes two arguments to calculate area, and the second takes three arguments to calculate area.

When an overloaded method is called java look for match between the arguments to call the method and the method's parameters. This match need not always be exact, sometime when exact match is not found, Java automatic type conversion plays a vital role.

---

### Example of Method overloading with type promotion.

```
class Area
{
    void find(long l,long b)
    {
        System.out.println("Area is"+(l*b)) ;
    }
    void find(int l, int b,int h)
    {
        System.out.println("Area is"+(l*b*h));
    }
    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(8,5);    //automatic type conversion from find(int,int) to find(long,
long) .
        ar.find(2,4,6)    //find(int l, int b,int h) is called.
    }
}
```

#### Output:

Area is 40

Area is 48

## Constructors in Java

A constructor is a special method that is used to initialize an object. Every class has a constructor, if we don't explicitly declare a constructor for any java class the compiler builds a default constructor for that class. A constructor does not have any return type.

A constructor has same name as the class in which it resides. Constructor in Java can not be abstract, static, final or synchronized. These modifiers are not allowed for constructor.

```
class Car
{
    String name ;
    String model;
    Car( )    //Constructor
    {
        name = "";
        model="";
    }
}
```

---

## There are two types of Constructor

- Default Constructor
- Parameterized constructor

Each time a new object is created at least one constructor will be invoked.

```
Car c = new Car()    //Default constructor invoked
Car c = new Car(name); //Parameterized constructor invoked
```

---

## Constructor Overloading

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is not much different than method overloading. In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading you have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java.

---

## Q. Why do we Overload constructors ?

Constructor overloading is done to construct object in different ways.

---

### Example of constructor overloading

```
class Cricketer
{
    String name;
    String team;
    int age;
    Cricketer ()    //default constructor.
    {
        name = "";
        team = "";
        age = 0;
    }
    Cricketer(String n, String t, int a)    //constructor overloaded
    {
        name = n;
        team = t;
        age = a;
    }
    Cricketer (Cricketer ckt)    //constructor similar to copy constructor of c++
    {
        name = ckt.name;
        team = ckt.team;
        age = ckt.age;
    }
    public String toString()
    {
```

```
    return "this is " + name + " of "+team;
}
}
```

Class test:

```
{
    public static void main (String[] args)
    {
        Cricketer c1 = new Cricketer();
        Cricketer c2 = new Cricketer("sachin", "India", 32);
        Cricketer c3 = new Cricketer(c2 );
        System.out.println(c2);
        System.out.println(c3);
        c1.name = "Virat";
        c1.team= "India";
        c1.age = 32;
        System .out. print in (c1);
    }
}
```

**output:**

```
this is sachin of india
this is sachin of india
this is virat of india
```

---

## **Q What's the difference between constructors and normal methods?**

Constructors must have the same name as the class and can not return a value. They are only called once while regular methods could be called many times and it can return a value or can be void.

---

### Q. What is constructor chaining in Java?

Constructor chaining is a phenomena of calling one constructor from another constructor of same class. Since constructor can only be called from another constructor in Java, constructor chaining is used for this purpose.

```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int x)
    {
        System.out.println("x="+x);
    }
}
```

---

### Q. Does constructors return any value?

Yes, constructors return current instant of a class. But yet constructor signature cannot have any return type.

## this keyword

- **this** keyword is used to refer to current object.
- **this** is always a reference to the object on which method was invoked.
- **this** can be used to invoke current class constructor.
- **this** can be passed as an argument to another method.

*Example :*

```
class Box
{
    Double width, weight, dept;
    Box (double w, double h, double d)
    {
        this.width = w;
        this.height = h;
        this.depth = d;
    }
}
```

Here the **this** is used to initialize member of current object.

---

### **The this is used to call overloaded constructor in java**

```
class Car
{
    private String name;
    public Car()
    {
        this("BMW");    //overloaded constructor is called.
    }
    public Car(Stting n)
    {
        this.name=n;    //member is initialized using this.
    }
}
```

---

**The this is also used to call Method of that class.**

```
public void getName()
{
    System.out.println("Studytonight");
}

public void display()
{
    this.getName();
    System.out.println();
}
```

---

**this is used to return current Object**

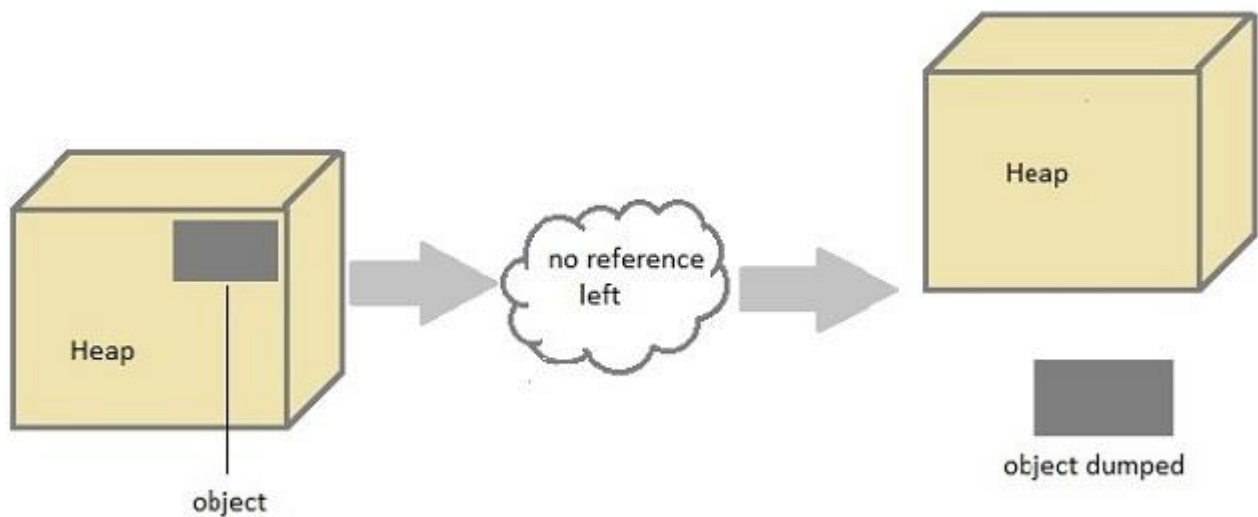
```
public Car getCar()
{
    return this;
}
```

## Garbage Collection

In Java destruction of object from memory is done automatically by the JVM. 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**. This is accomplished by the JVM.

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





---

### Can the Garbage Collection be forced explicitly ?

No, the Garbage Collection can not be forced explicitly. We may request JVM for **garbage collection** by calling **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.

---

### 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. **finalize()** method is called by garbage collection thread before collecting object. Its 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. `finalize()` method is defined in **java.lang.Object** class, therefore it is available to all the classes.
  2. `finalize()` method is declare as **protected** inside Object class.
  3. `finalize()` method gets called only once by GC threads.
- 

## gc() Method

**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 for 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");  
    }  
}
```

Output :

Garbage Collected

## Modifiers in Java

Modifiers are keywords that are added to change meaning of a definition. In Java, modifiers are categorized into two types,

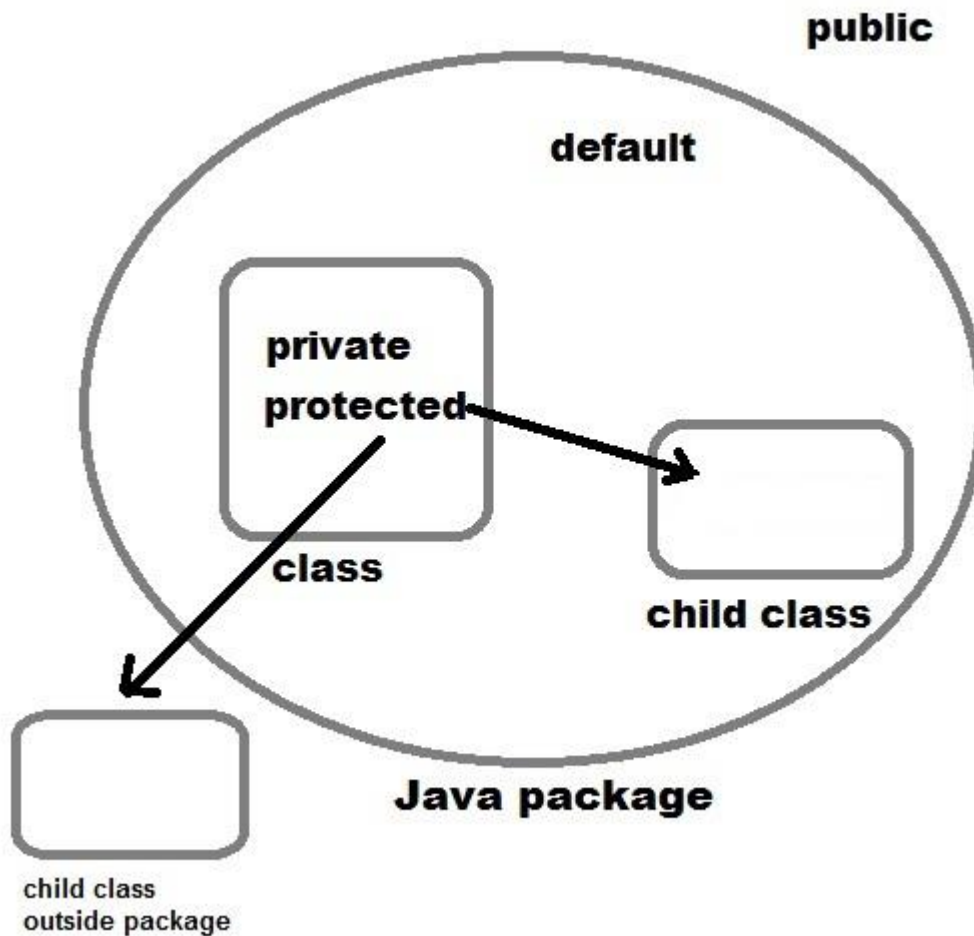
1. Access control modifier
2. Non Access Modifier

---

### 1) Access control modifier

Java language has four access modifier to control access levels for classes, variable methods and constructor.

- **Default** : Default has scope only inside the same package
- **Public** : Public scope is visible everywhere
- **Protected** : Protected has scope within the package and all sub classes
- **Private** : Private has scope only within the classes



---

## 2) Non-access Modifier

Non-access modifiers do not change the accessibility of variables and methods, but they do provide them special properties. Non-access modifiers are of 5 types,

1. Final
2. Static
3. Transient
4. Synchronized
5. Volatile

## Final

Final modifier is used to declare a field as final i.e. it prevents its content from being modified. Final field must be initialized when it is declared.

*Example :*

```
class Cloth
{
    final int MAX_PRICE = 999;    //final variable
    final int MIN_PRICE = 699;
    final void display()          //final method
    {
        System.out.println("Maxprice is" + MAX_PRICE );
        System.out.println("Minprice is" + MIN_PRICE);
    }
}
```

A class can also be declared as final. A class declared as final cannot be inherited. **String** class in java.lang package is an example of final class. Method declared as final can be inherited but you cannot override(redefine) it.

---

## Static Modifier

Static Modifiers are used to create class variable and class methods which can be accessed without instance of a class. Let's study how it works with variables and member functions.

### Static with Variables

Static variables are defined as a class member that can be accessed without any object of that class. Static variable has only one single storage. All the objects of the class having static variable will have the same instance of static variable. Static variables are initialized only once.

Static variables are used to represent common property of a class. It saves memory. Suppose there are 100 employees in a company. All employees have their unique name and employee ID but company

name will be same all 100 employee. Here company name is the common property. So if you create a class to store employee detail, company\_name field will be mark as static.

### Example

```
class Employee
{
    int e_id;
    String name;
    static String company_name = "StudyTonight";
}
```

---

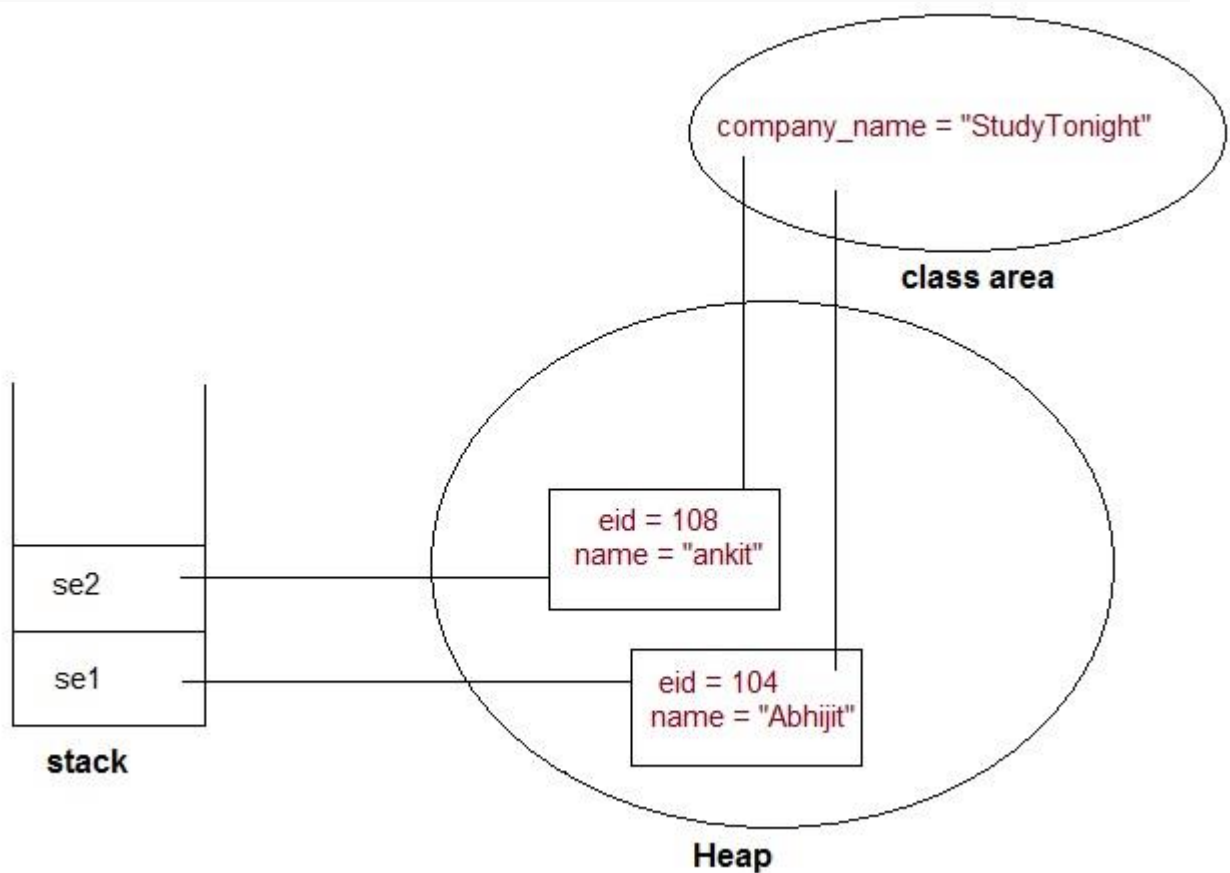
### Example of static variable

```
class ST_Employee
{
    int eid;
    String name;
    static String company_name ="StudyTonight";
    public void show()
    {
        System.out.println(eid+" "+name+" "+company_name);
    }
    public static void main( String[] args )
    {
        ST_Employee se1 = new ST_Employee();
        se1.eid = 104;
        se1.name = "Abhijit";
        se1.show();
        ST_Employee se2 = new ST_Employee();
        se2.eid = 108;
```

```
    se2.name = "ankit";  
    se2.show();  
}  
  
}
```

### Output

```
104 Abhijit StudyTonight  
108 ankit StudyTonight
```



---

### Static variable vs Instance Variable

Static variable	Instance Variable
Represent common property	Represent unique property
Accessed using class name	Accessed using object
get memory only once	get new memory each time a new object is created

### Example

```

public class Test
{
    static int x = 100;
    int y = 100;
    public void increment()
    {
        x++; y++;
    }
    public static void main( String[] args )
    {
        Test t1 = new Test();
        Test t2 = new Test();
        t1.increment();
        t2.increment();
        System.out.println(t2.y);
        System.out.println(Test.x); //accessed without any instance of class.
    }
}

```

### Output



101

102

See the difference in value of two variable. Static variable **y** shows the changes made to it by increment() method on the different object. While instance variable **x** show only the change made to it by increment() method on that particular instance.

---

## Static Method

A method can also be declared as static. Static methods do not need instance of its class for being accessed. main() method is the most common example of static method. main() method is declared as static because it is called before any object of the class is created.

*Example :*

```
class Test
{

    public static void square(int x)
    {
        System.out.println(x*x);
    }

    public static void main (String[] arg)
    {

        square(8)    //static method square () is called without any instance of class.
    }
}
```

**Output: 64**

---

## Static block

Static block is used to initialize static data member. Static block executes before `main()` method.

### Example

```
class ST_Employee
{
    int eid;
    String name;
    static String company_name;

    static {
        company_name = "StudyTonight";    //static block invoked before main() method
    }

    public void show()
    {
        System.out.println(eid+" "+name+" "+company_name);
    }

    public static void main( String[] args )
    {
        ST_Employee se1 = new ST_Employee();
        se1.eid = 104;
        se1.name = "Abhijit";
        se1.show();

    }
}
```

### Output

```
104 Abhijit StudyTonight
```

---

## Q. Why a non-static variable cannot be referenced from a static context ?

When you try to access a non-static variable from a static context like main method, java compiler throws a message like *"a non-static variable cannot be referenced from a static context"*. This is because non-static variables are related with instance of class(object) and they get created when instance of a class is created by using **new** operator. So if you try to access a non-static variable without any instance compiler will complain because those variables are not yet created and they don't have any existence until an instance is created and associated with it.

### Example of accessing non-static variable from a static context

```
class Test
{
    int x;
    public static void main(String[] args)
    {
        x=10;
    }
}
```

### Output

compiler error: non-static variable count cannot be referenced from a static context

### Same example using instance of class

```
class Test
{
    int x;
    public static void main(String[] args)
    {
        Test tt=new Test();
        tt.x=10; //works fine with instance of class
    }
}
```

```
}
```

---

### Q. Why main() method is static in java ?

Because static methods can be called without any instance of a class and **main()** is called before any instance of a class is created.

---

### Transient modifier

When an instance variable is declared as transient, then its value doesn't persist when an object is serialized

---

### Synchronized modifier

When a method is synchronized it can be accessed by only one thread at a time. We will discuss it in detail in Thread.

---

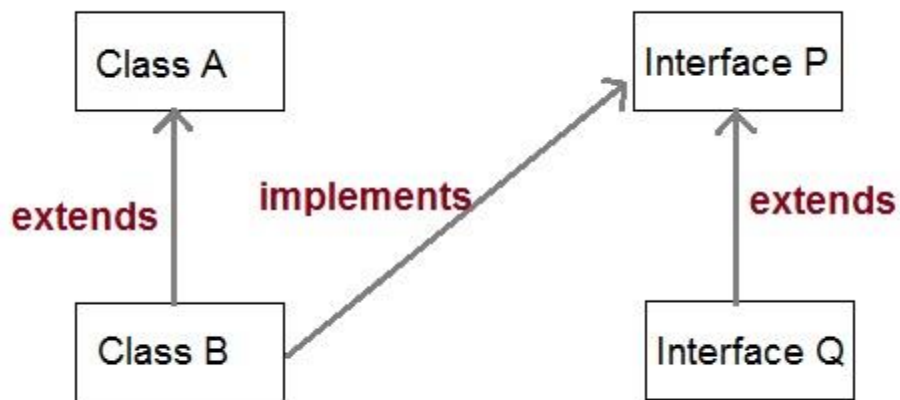
### Volatile modifier

Volatile modifier tells the compiler that the volatile variable can be changed unexpectedly by other parts of your program. Volatile variables are used in case of multithreading program.

## Inheritance (IS-A)

Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed **a class to inherit property of another class**. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child) in Java language.

Inheritance defines **is-a** relationship between a Super class and its Sub class. **extends** and **implements** keywords are used to describe inheritance in Java.



Let us see how **extend** keyword is used to achieve Inheritance.

```
class Vehicle.  
{  
    .....  
}  
class Car extends Vehicle  
{  
    ..... //extends the property of vehicle class.  
}
```

Now based on above example. In OOPs term we can say that,

- **Vehicle** is super class of **Car**.
- **Car** is sub class of **Vehicle**.
- Car IS-A Vehicle.

---

## Purpose of Inheritance

1. To promote code reuse.
2. To use Polymorphism.

---

## Simple example of Inheritance

```
class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}

public class Child extends Parent {
    public void c1()
    {
        System.out.println("Child method");
    }

    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.c1();    //method of Child class
        cobj.p1();    //method of Parent class
    }
}
```

### Output

```
Child method
Parent method
```

---

## Another example of Inheritance

```
class Vehicle
{
```

```
String vehicleType;
}
public class Car extends Vehicle {

    String modelType;
    public void showDetail()
    {
        vehicleType = "Car";           //accessing Vehicle class member
        modelType = "sports";
        System.out.println(modelType+" "+vehicleType);
    }
    public static void main(String[] args)
    {
        Car car =new Car();
        car.showDetail();
    }
}
```

### Output

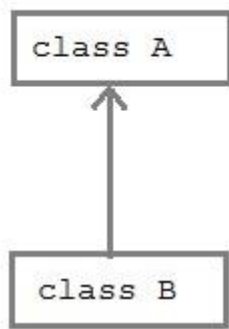
sports Car

---

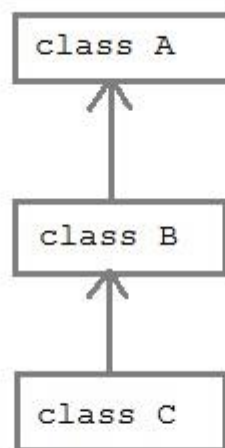
## Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Heirarchical Inheritance

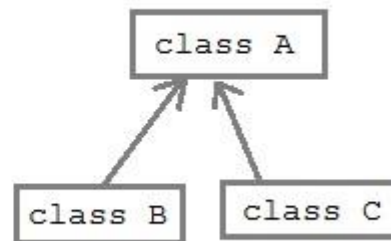
**NOTE :**Multiple inheritance is not supported in java



**Simple Inheritance**



**Multilevel inheritance**



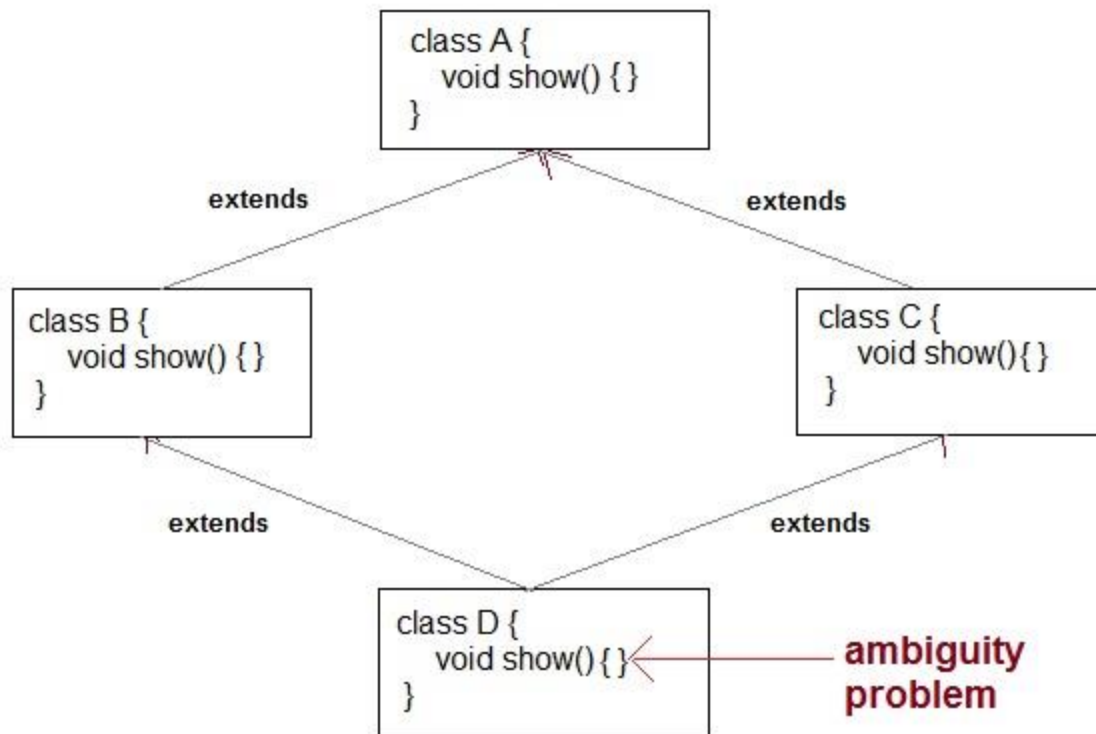
**Heirarchical inheritance**

---

### Why multiple inheritance is not supported in Java

- To remove ambiguity.
- To provide more maintainable and clear design.





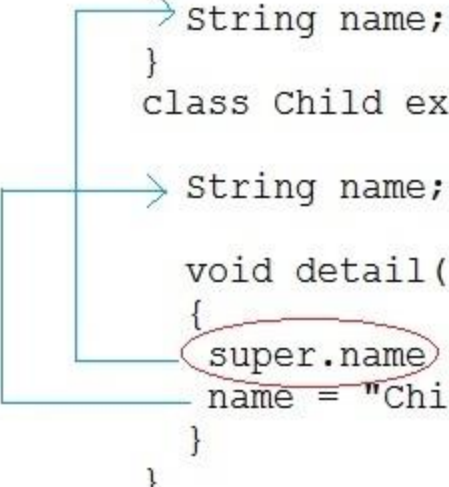
---

## super keyword

In Java, `super` keyword is used to refer to immediate parent class of a class. In other words **super** keyword is used by a subclass whenever it need to refer to its immediate super class.

```
class Parent
{
    String name;
}
class Child extends Parent {
    String name;

    void detail()
    {
        super.name = "Parent";
        name = "Child";
    }
}
```



---

### Example of Child class referring Parent class **property** using **super** keyword

```
class Parent
{
    String name;
}

public class Child extends Parent {
    String name;

    public void details()
    {
        super.name = "Parent";    //refers to parent class member
        name = "Child";

        System.out.println(super.name+" and "+name);
    }

    public static void main(String[] args)
```

```
{
    Child cobj = new Child();
    cobj.details();
}
}
```

### Output

Parent and Child

---

### Example of Child class refering Parent class methods using super keyword

```
class Parent
{
    String name;
    public void details()
    {
        name = "Parent";
        System.out.println(name);
    }
}

public class Child extends Parent {
    String name;
    public void details()
    {
        super.details();    //calling Parent class details() method
        name = "Child";
        System.out.println(name);
    }
    public static void main(String[] args)
    {
```

```
        Child cobj = new Child();
        cobj.details();
    }
}
```

### Output

Parent

Child

---

### Example of Child class calling Parent class constructor using super keyword

```
class Parent
{
    String name;

    public Parent(String n)
    {
        name = n;
    }
}

public class Child extends Parent {
    String name;

    public Child(String n1, String n2)
    {

        super(n1);        //passing argument to parent class constructor
        this.name = n2;
    }
}
```

```
}  
  
public void details()  
{  
    System.out.println(super.name+" and "+name);  
}  
  
public static void main(String[] args)  
{  
    Child cobj = new Child("Parent","Child");  
    cobj.details();  
}  
}
```

### Output

Parent and Child

---

### Super class reference pointing to Sub class object.

In context to above example where Class B extends class A.

```
A a=new B();
```

is legal syntax because of IS-A relationship is there between class A and Class B.

---

### Q. Can you use both this() and super() in a Constructor?

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

## Aggregation (HAS-A)

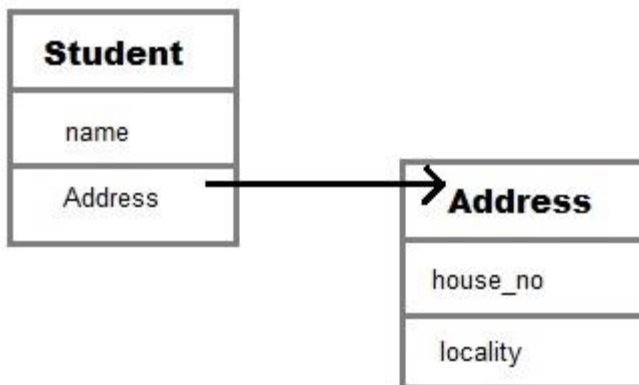
HAS-A relationship is based on usage, rather than inheritance. In other words, class A *has-*a relationship with class B, if code in class A has a reference to an instance of class B.

---

## Example

```
class Student
{
    String name;
    Address ad;
}
```

Here you can say that **Student** has-a **Address**.



**Student** class has an instance variable of type Address. Student code can use Address reference to invoke methods on the **Address**, and get **Address** behavior.

Aggregation allow you to design classes that follow good Object Oriented practices. It also provide code reusability.

---

## Example of Aggregation

```
class Author
{
    String authorName;
    int age;
    String place;
    Author(String name,int age,String place)
```

```
{
    this.authorName=name;
    this.age=age;
    this.place=place;
}
public String getAuthorName()
{
    return authorName;
}
public int getAge()
{
    return age;
}
public String getPlace()
{
    return place;
}
}

class Book
{
    String name;
    int price;
    Author auth;
    Book(String n,int p,Author at)
    {
        this.name=n;
        this.price=p;
        this.auth=at;
    }
}
```

```
}  
public void showDetail()  
{  
    System.out.println("Book is"+name);  
    System.out.println("price "+price);  
    System.out.println("Author is "+auth.getAuthorName());  
}  
}
```

```
class Test  
{  
    public static void main(String args[])  
    {  
        Author ath=new Author("Me",22,"India");  
        Book b=new Book("Java",550,ath);  
        b.showDetail();  
    }  
}
```

#### Output:

Book is Java.

price is 550.

Author is me.

---

## Q. What is Composition in java?

Composition is restricted form of Aggregation. For example a class **Car** cannot exist without **Engine**.

```
class Car  
{  
    private Engine engine;
```



```
Car(Engine en)
{
    engine = en;
}
}
```

---

### Q. When to use Inheritance and Aggregation?

When you need to use property and behaviour of a class without modifying it inside your class. In such case **Aggregation** is a better option. Whereas when you need to use and modify property and behaviour of a class inside your class, its best to use **Inheritance**.

## Method Overriding

When a method in a sub class has same name and type signature as a method in its super class, then the method is known as overridden method. Method overriding is also referred to as runtime polymorphism. The key benefit of overriding is the ability to **define method that's specific to a particular subclass type**.

---

### Example of Method Overriding

```
class Animal
{
    public void eat()
    {
        System.out.println("Generic Animal eating");
    }
}

class Dog extends Animal
```

```

{
    public void eat()    //eat() method overridden by Dog class.
    {
        System.out.println("Dog eat meat");
    }
}

```

As you can see here Dog class gives it own implementation of eat() method. Method must have same name and same type signature.

**NOTE :** Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

---

## Covariant return type

Since Java 5, it is possible to override a method by changing its return type. If subclass override any method by changing the return type of super class method, then the return type of overridden method must be **subtype of return type** declared in original method inside the super class. This is the only way by which method can be overridden by changing its return type.

*Example :*

```

class Animal
{
    Animal myType()
    {
        return new Animal();
    }
}

class Dog extends Animal
{
    Dog myType()    //Legal override after Java5 onward
    {

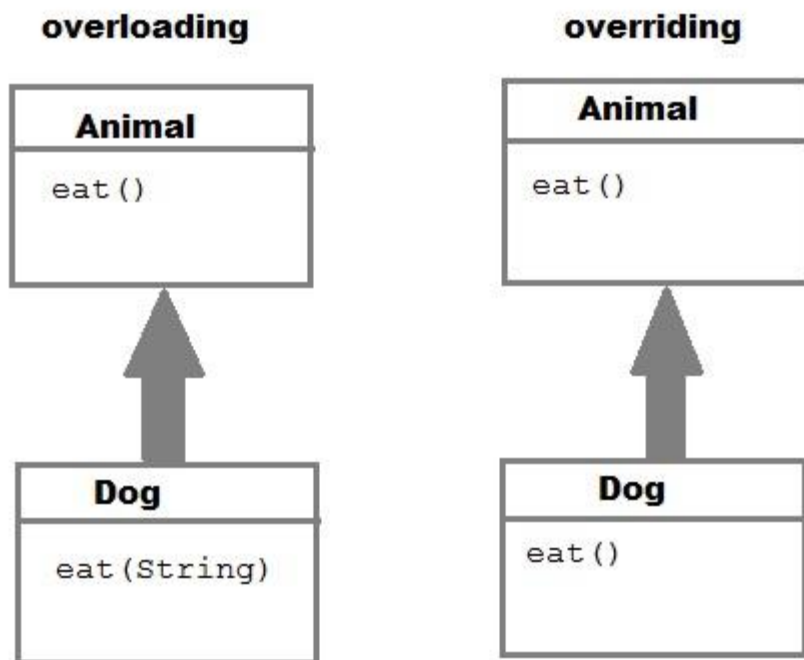
```

```
return new Dog();  
}  
}
```

---

## Difference between Overloading and Overriding

Method Overloading	Method Overriding
Parameter must be different and name must be same.	Both name and parameter must be same.
Compile time polymorphism.	Runtime polymorphism.
Increase readability of code.	Increase reusability of code.
Access specifier can be changed.	Access specifier must not be more restrictive than original method(can be less restrictive).



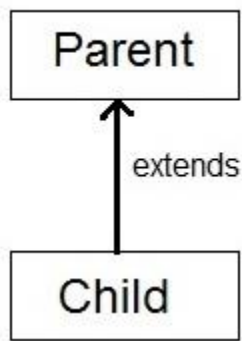
---

### Q. Can we Override static method ? Explain with reasons ?

No, we cannot override static method. Because static method is bound to class whereas method overriding is associated with object i.e at runtime.

## Runtime Polymorphism or Dynamic method dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );

←  
**Upcasting**

~~Child c = new Parent( );~~

**incompatible type**

---

## Upcasting

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**

---

## Example

```
class Game
{
    public void type()
    { System.out.println("Indoor & outdoor"); }
}
```

Class **Cricket** extends **Game**

```
{
    public void type()
    { System.out.println("outdoor game"); }
```

```

public static void main(String[] args)
{
    Game gm = new Game();
    Cricket ck = new Cricket();
    gm.type();
    ck.type();
    gm=ck;      //gm refers to Cricket object
    gm.type();  //calls Cricket's version of type
}
}

```

#### Output:

Indoor & outdoor

Outdoor game

Outdoor game

Notice the last output. This is because of **gm = ck**; Now `gm.type()` will call Cricket version of type method. Because here gm refers to cricket object.

### Q. Difference between Static binding and Dynamic binding in java ?

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

## instanceof operator

In Java, `instanceof` operator is used to check the type of an object at runtime. It is the means by which your program can obtain run-time type information about an object. `instanceof` operator is

also important in case of casting object at runtime. `instanceof` operator return boolean value, if an object reference is of specified type then it return **true** otherwise **false**.

---

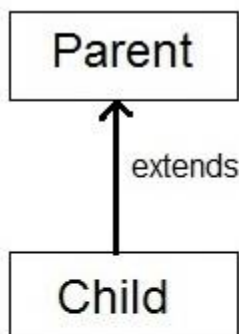
### Example of `instanceof`

```
public class Test
{
    public static void main(String[] args)
    {
        Test t= new Test();
        System.out.println(t instanceof Test);
    }
}
```

output `true`

---

### Downcasting



Parent p = new Child( );

Upcasting

~~Child c = new Parent( );~~

Compile time error

Child c = ( Child ) new Parent( );

Downcasting but throws  
ClassCastException at runtime.

---

## Example of downcasting with instanceof operator

```
class Parent{ }

public class Child extends Parent
{
    public void check()
    {
        System.out.println("Sucessfull Casting");
    }

    public static void show(Parent p)
    {
        if(p instanceof Child)
        {
            Child b1=(Child)p;
            b1.check();
        }
    }

    public static void main(String[] args)
    {
        Parent p=new Child();

        Child.show(p);

    }
}
```



## Output

Sucessfull Casting

---

### More example of instanceof operator

```
class Parent{}

class Child1 extends Parent{}

class Child2 extends Parent{}

class Test
{
    public static void main(String[] args)
    {
        Parent p =new Parent();
        Child1 c1 = new Child1();
        Child2 c2 = new Child2();

        System.out.println(c1 instanceof Parent);           //true
        System.out.println(c2 instanceof Parent);           //true
        System.out.println(p instanceof Child1);            //false
        System.out.println(p instanceof Child2);            //false

        p = c1;
        System.out.println(p instanceof Child1);            //true
        System.out.println(p instanceof Child2);            //false

        p = c2;
```

```

        System.out.println(p instanceof Child1);           //false
        System.out.println(p instanceof Child2);           //true

    }

}

```

### Output

```

true
true
false
false
true
false
false
true

```

## Command line argument in Java

The command line argument is the argument passed to a program at the time when you run it. To access the command-line argument inside a java program is quite easy, they are stored as string in **String** array passed to the args parameter of `main()` method.

### Example

```

class cmd
{
    public static void main(String[] args)
    {
        for(int i=0;i< args.length;i++)
        {
            System.out.println(args[i]);
        }
    }
}

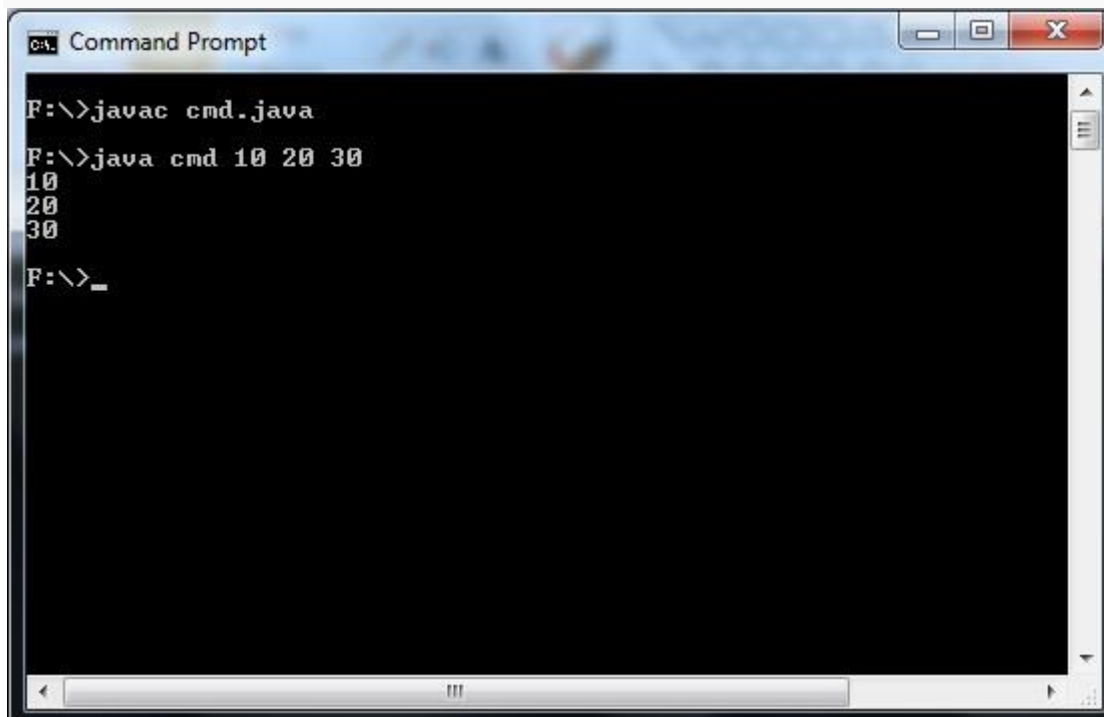
```

```
}  
}  
}
```

Execute this program as `java cmd 10 20 30`

### Output

```
10  
20  
30
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a black background with white text. The text inside the window shows the following commands and output:

```
F:\>javac cmd.java  
F:\>java cmd 10 20 30  
10  
20  
30  
F:\>_
```

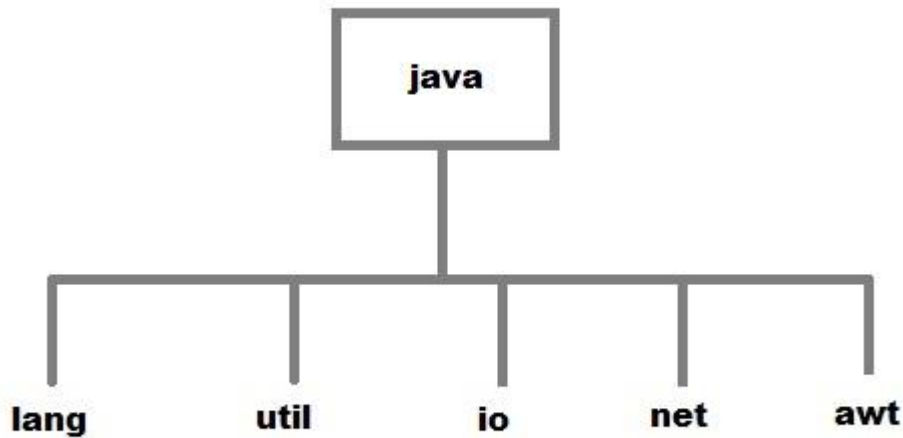
## Java Package

Package are used in Java, in-order to avoid name conflicts and to control access of class, interface and enumeration etc. A package can be defined as a group of similar types of classes, interface, enumeration and sub-package. Using package it becomes easier to locate the related classes.

---

**Package are categorized into two forms**

- Built-in Package:-Existing Java package for example `java.lang`, `java.util` etc.
- User-defined-package:- Java package created by user to categorized classes and interface



---

## Creating a package

Creating a package in java is quite easy. Simply include a package command followed by name of the package as the first statement in java source file.

```
package mypack;  
public class employee  
{  
    ...statement;  
}
```

The above statement create a package called **mypack**.

Java uses file system directory to store package. For example the `.class` for any classes you to define to be part of **mypack** package must be stored in a directory called mypack

---

## Example of package creation

```
package mypack

class Book
{
    String bookname;
    String author;
    Book(String b, String c)
    {
        this.bookname = b;
        this.author = c;
    }
    public void show()
    {
        System.out.println(bookname+" "+ author);
    }
}

class test
{
    public static void main(String[] args)
    {
        Book bk = new Book("java","Herbert");
        bk.show();
    }
}
```

To run this program :

- create a directory under your current working development directory(i.e. JDK directory), name it **asmypack**.

- compile the source file
- Put the class file into the directory you have created.
- Execute the program from development directory.

**NOTE :** Development directory is the directory where your JDK is install.

---

## Uses of java package

Package is a way to organize files in java, it is used when a project consists of multiple modules. It also helps resolve naming conflicts. Package's access level also allows you to protect data from being used by the non-authorized classes.

---

## import keyword

**import** keyword is used to import built-in and user-defined packages into your java source file. So that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to class that is present in different package

1. **Using fully qualified name** (But this is not a good practice.)

*Example :*

```
class MyDate extends java.util.Date
{
    //statement;
}
```

2. **import the only class you want to use.**

*Example :*

```
import java.util.Date;
class MyDate extends Date
{
```

```
//statement.  
}
```

### 3. import all the classes from the particular package

*Example :*

```
import java.util.*;  
class MyDate extends Date  
{  
    //statement;  
}
```

---

**import statement is kept after the package statement.**

*Example :*

```
package mypack;  
import java.util.*;
```

But if you are not creating any package then import statement will be the first statement of your java source file.

---

## Static import

**static import** is a feature that expands the capabilities of **import** keyword. It is used to import **static** member of a class. We all know that static member are referred in association with its class name outside the class. Using **static import**, it is possible to refer to the static member directly without its class name. There are two general form of static import statement.

- The first form of **static import** statement, import only a single static member of a class

### Syntax

```
import static package.class-name.static-member-name;
```

### Example

```
import static java.lang.Math.sqrt;    //importing static method sqrt of Math c
lass
```

- The second form of **static import** statement, imports all the static member of a class

### Syntax

```
import static package.class-type-name.*;
```

### Example

```
import static java.lang.Math.*;        //importing all static member of Math c
lass
```

---

## Example without using static import

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(144));
    }
}
```

### Output

12

---

## Example using static import

```
import static java.lang.Math.*;
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(sqrt(144));
    }
}
```



```
    }  
}
```

### Output

```
12
```

## Abstract class

If a class contain any abstract method then the class is declared as abstract class. An abstract class is never instantiated. It is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method.

### Syntax :

```
abstract class class_name { }
```

---

### Abstract method

Method that are declared without any body within an abstract class is known as abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

### Syntax :

```
abstract return_type function_name ();    // No definition
```

---

### Example of Abstract class

```
abstract class A  
{  
    abstract void callme();  
}  
  
class B extends A  
{
```

```
void callme()
{
    System.out.println("this is callme.");
}
public static void main(String[] args)
{
    B b=new B();
    b.callme();
}
}
```

**output:** this is callme.

---

### **Abstract class with concrete(normal) method.**

Abstract classes can also have normal methods with definitions, along with abstract methods.

```
abstract class A
{
    abstract void callme();
    public void normal()
    {
        System.out.println("this is concrete method");
    }
}
class B extends A
{
    void callme()
    {
        System.out.println("this is callme.");
    }
}
```

```
public static void main(String[] args)
{
    B b=new B();
    b.callme();
    b.normal();
}
}
```

**output:**

this is callme.

this is concrete method.

---

## Points to Remember

1. Abstract classes are not Interfaces. They are different, we will study this when we will study Interfaces.
2. An abstract class must have an abstract method.
3. Abstract classes can have Constructors, Member variables and Normal methods.
4. Abstract classes are never instantiated.
5. When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

---

## Abstraction using abstract class

Abstraction is an important feature of OOPS. It means hiding complexity. Abstract class is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method. Lets see how abstract class is used to provide abstraction.

```
abstract class Vehicle
{
    public abstract void engine();
}
```

```
}  
  
public class Car extends Vehicle {  
  
    public void engine()  
    {  
        System.out.println("Car engine");  
        //car engine implementation  
    }  
  
    public static void main(String[] args)  
    {  
        Vehicle v = new Car();  
        v.engine();  
  
    }  
}
```

### Output

Car engine

Here by casting instance of **Car** type to **Vehicle** reference, we are hiding the complexity of **Car** type under **Vehicle**. Now the **Vehicle** reference can be used to provide the implementation but it will hide the actual implementation process.

---

## When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

# Interface

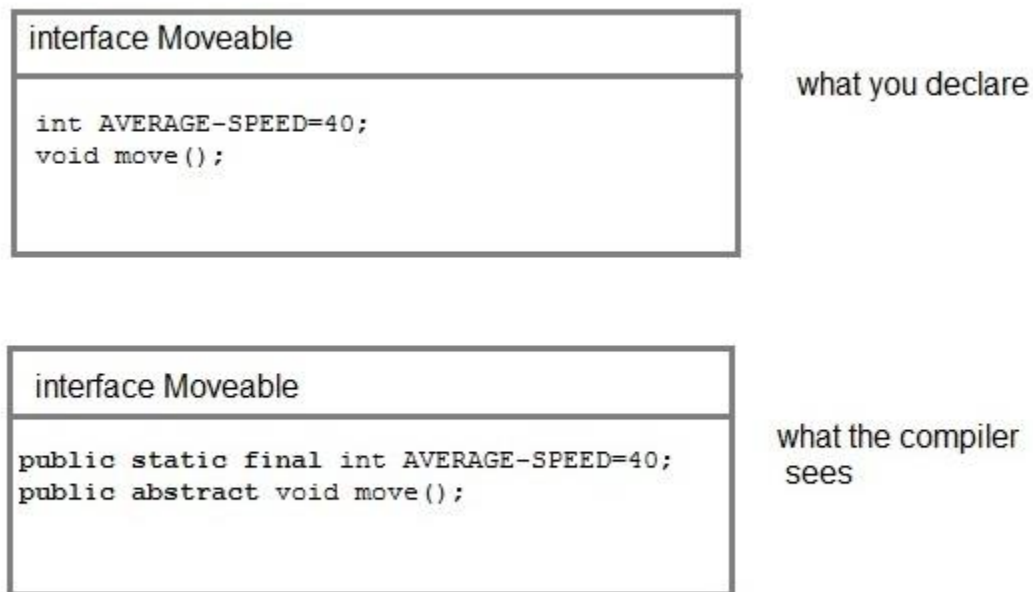
Interface is a pure abstract class. They are syntactically similar to classes, but you cannot create instance of an **Interface** and their methods are declared without any body. Interface is used to achieve complete **abstraction** in Java. When you create an interface it defines what a class can do without saying anything about how the class will do it.

## Syntax :

```
interface interface_name { }
```

## Example of Interface

```
interface Moveable
{
    int AVERAGE-SPEED=40;
    void move();
}
```



**NOTE :** Compiler automatically converts methods of Interface as public and abstract, and the data members as public, static and final by default.

---

## Rules for using Interface

- Methods inside Interface must not be static, final, native or strictfp.
- All variables declared inside interface are implicitly public static final variables(constants).
- All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.

---

## Example of Interface implementation

```
interface Moveable
{
    int AVG-SPEED = 40;
    void move();
}

class Vehicle implements Moveable
{
    public void move()
    {
        System.out.println("Average speed is"+AVG-SPEED);
    }
    public static void main (String[] arg)
    {
        Vehicle vc = new Vehicle();
        vc.move();
    }
}
```

```
}  
}
```

**Output:**

Average speed is 40.

---

## Interfaces supports Multiple Inheritance

Though classes in java doesn't support multiple inheritance, but a class can implement more than one interface.

```
interface Moveable  
{  
    boolean isMoveable();  
}
```

```
interface Rollable  
{  
    boolean isRollable  
}
```

```
class Tyre implements Moveable, Rollable  
{  
    int width;  
  
    boolean isMoveable()  
    {  
        return true;  
    }  
  
    boolean isRollable()
```

```
{
    return true;
}
public static void main(String args[])
{
    Tyre tr=new Tyre();
    System.out.println(tr.isMoveable());
    System.out.println(tr.isRollable());
}
}
```

**Output:**

true

true

---

## Interface extends other Interface

Classes implements interfaces, but an interface extends other interface.

```
interface NewsPaper
{
    news();
}

interface Magazine extends NewsPaper
{
    colorful();
}
```

---



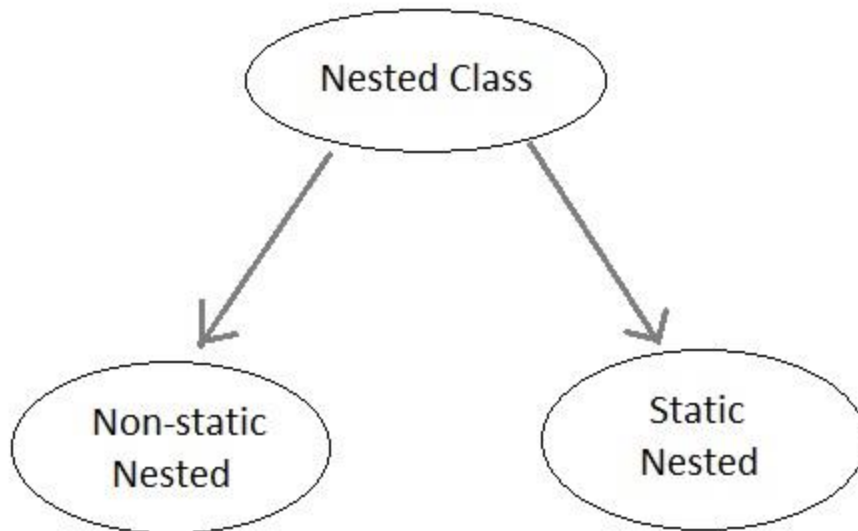
## Difference between an interface and an abstract class?

Abstract class	Interface
Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classes.	Interface is a Java Object containing method declaration but no implementation. The classes which implement the Interfaces must provide the method definition for all the methods.
Abstract class is a Class prefix with an abstract keyword followed by Class definition.	Interface is a pure abstract class which starts with interface keyword.
Abstract class can also contain concrete methods.	Whereas, Interface contains all abstract methods and final variable declarations.
Abstract classes are useful in a situation that Some general methods should be implemented and specialization behavior should be implemented by child classes.	Interfaces are useful in a situation that all properties should be implemented.

---

# Nested Class

A class within another class is known as Nested class. The scope of the nested is bounded by the scope of its enclosing class.



---

## Static Nested Class

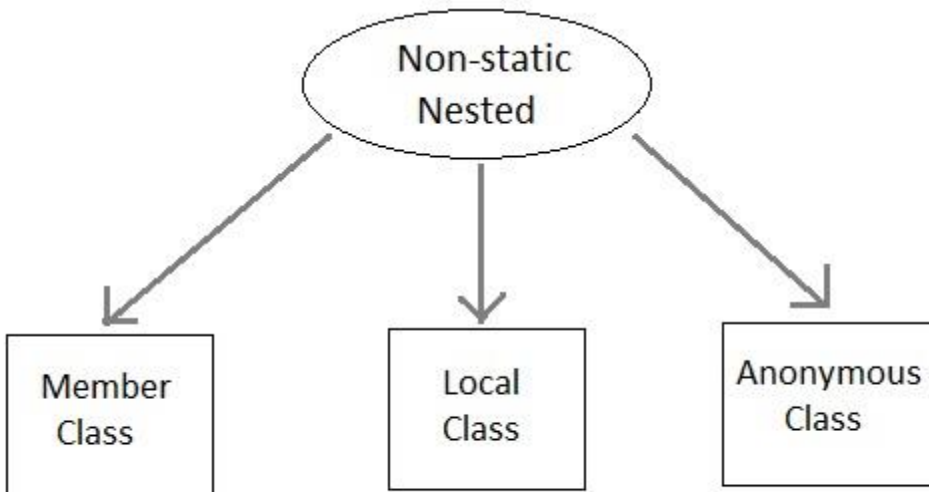
A static nested class is the one that has **static** modifier applied. Because it is static it cannot refer to non-static members of its enclosing class directly. Because of this restriction static nested class is seldom used.

---

## Non-static Nested class

Non-static Nested class is most important type of nested class. It is also known as **Inner** class. It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class.

One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.



---

### Example of Inner class

```
class Outer
{
    public void display()
    {
        Inner in=new Inner();
        in.show();
    }

    class Inner
    {
        public void show()
        {
            System.out.println("Inside inner");
        }
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        Outer ot=new Outer();
        ot.display();
    }
}
```

**Output:**

Inside inner

---

### Example of Inner class inside a method

```
class Outer
{
    int count;
    public void display()
    {
        for(int i=0;i<5;i++)
        {
            class Inner        //Inner class defined inside for loop
            {
                public void show()
                {
                    System.out.println("Inside inner "+(count++));
                }
            }
            Inner in=new Inner();
        }
    }
}
```

```
        in.show();
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer ot=new Outer();
        ot.display();
    }
}
```

**Output:**

```
Inside inner 0
Inside inner 1
Inside inner 2
Inside inner 3
Inside inner 4
```

---

### Example of Inner class instantiated outside Outer class

```
class Outer
{
    int count;
    public void display()
    {
        Inner in=new Inner();
        in.show();
    }
}
```

```

}

class Inner
{
    public void show()
    {
        System.out.println("Inside inner "+(++count));
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer ot=new Outer();
        Outer.Inner in= ot.new Inner();
        in.show();
    }
}

```

### Output

Inside inner 1

---

## Anonymous class

A class without any name is called Anonymous class.

```

interface Animal
{
    void type();
}

```

```
}  
public class ATest {  
    public static void main(String args[])  
    {  
        Animal an = new Animal(){           //Anonymous class created  
        public void type()  
        {  
            System.out.println("Anonymous animal");  
        }  
    };  
    an.type();  
}  
}
```

#### Output

Anonymous animal

Here a class is created which implements **Animal** interface and its name will be decided by the compiler. This anonymous class will provide implementation of **type()** method.