

Java OOPS

What is OOPS

OOPS is a programming approach which provides solution to real life 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 languages such as C, ALGOL, PASCAL etc. [Click here to watch video on OOPS concept in Java](#)

Java is an object oriented language which supports **object oriented concepts like: class and object**. In OOPS data is treated important and **encapsulated within the class**, object then use to access that data during runtime.

OOPS provides advantages over the other programming paradigm and include following features.

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.

Java Class

In Java everything is **encapsulated under classes**. Class is the core of Java language. It can be defined as **a template** that describe the behaviors and 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.

In Java, to declare a class **class** keyword is used. A class contain both **data and methods** 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**.

Thus, the instance variables and methods are known as **class members**.

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 only one parent class. By default, it extends **Object** class.
- The variables and methods are declared within a set of curly braces.

A Java class can contains fields, methods, constructors, and blocks. Lets see a general structure of a class.

Java class Syntax

```
class class_name {  
  
    field;  
  
    method;  
  
}
```

Copy

A simple class example

Suppose, **Student** is a class and student's name, roll number, age are its **fields** and **info()** is a **method**. Then class will look like below.

```
class Student.  
  
{  
  
    String name;  
  
    int rollno;  
  
    int age;  
  
void info(){  
  
    // some code  
  
}  
  
}
```

Copy

This is how a class look structurally. It is a blueprint for an object. We can call its fields and methods by using the object.

The fields declared inside the class are known as **instance variables**. It gets memory when an object is created at runtime.

Methods in the class are similar to the functions that are used to perform operations and represent behavior of an object.

After learning about the class, now lets understand what is an object.

Java Object

Object is an instance of a class while class is a blueprint of an object. An object represents the class and consists of **properties** and **behavior**.

Properties refer to the fields declared with in class and behavior represents to the methods available in the class.

In real world, we can understand object as a cell phone that has its properties like: name, cost, color etc and behavior like calling, chatting etc.

So we can say that object is a real world entity. Some real world objects are: ball, fan, car etc.

There is a syntax to create an object in the Java.

Java Object Syntax

```
className variable_name = new className();
```

Copy

Here, **className** is the name of class that can be anything like: Student that we declared in the above example.

variable_name is name of reference variable that is used to **hold the reference** of created object.

The **new** is a keyword which is used to allocate memory for the object.

Lets see an example to create an object of class Student that we created in the above class section.

Although there are many other ways by which we can create object of the class. we have covered this section in details in a [separate topics](#).

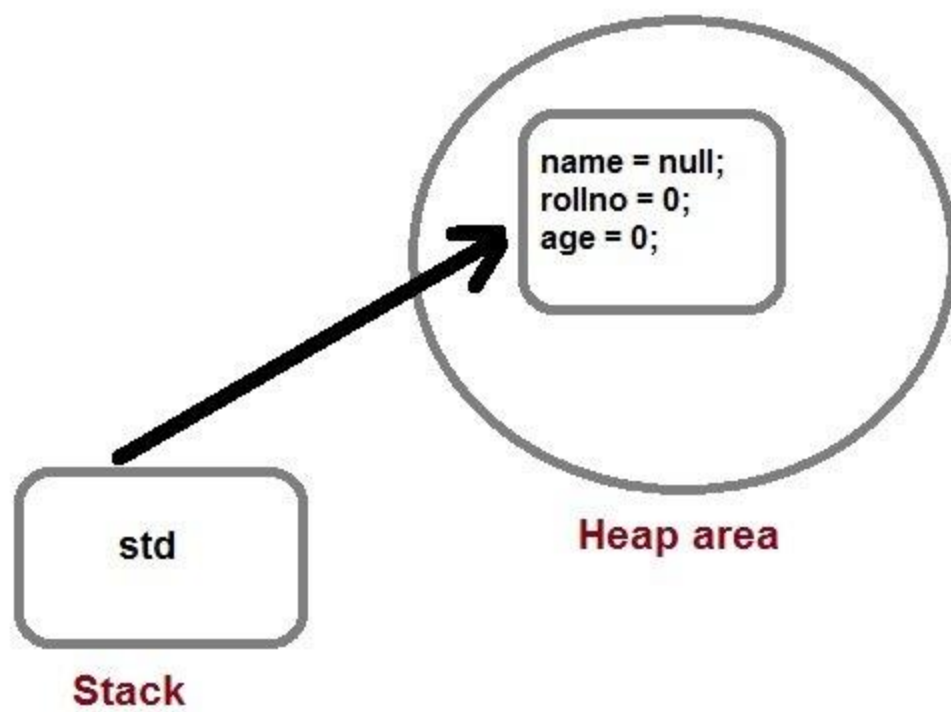
Example: Object creation

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

Copy

Here, **std** is an object that represents the class Student during runtime.

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.**



In this image, we can get idea how the an object refer to the memory area.

Now lets understand object and class combinally by using a real example.

Example: Creating a Class and its object

```
public class Student{

    String name;

    int rollno;

    int age;

    void info(){

        System.out.println("Name: "+name);

        System.out.println("Roll Number: "+rollno);

        System.out.println("Age: "+age);

    }

    public static void main(String[] args) {

        Student student = new Student();

    }

}
```

```
        // Accessing and property value

        student.name = "Ramesh";

        student.rollno = 253;

        student.age = 25;


        // Calling method

        student.info();

    }

}
```

Copy

Name: Ramesh

Roll Number: 253

Age: 25

In this example, we created a class Student and an object. Here you may be surprised of seeing main() method but don't worry it is just an entry point of the program by which JVM starts execution.

Here, we used main method to create object of Student class and access its fields and methods.

Example: Class fields

```
public class Student{

    String name;

    int rollno;

    int age;


    void info(){

        System.out.println("Name: "+name);

    }

}
```

```
        System.out.println("Roll Number: "+rollno);

        System.out.println("Age: "+age);

    }

    public static void main(String[] args) {

        Student student = new Student();

        // Calling method

        student.info();

    }

}
```

Copy

Name: null

Roll Number: 0

Age: 0

In case, if we don't initialized values of class fields then they are initialized with their **default values**.

Default values of instance variables

int, byte, short, long -> 0

float, double → 0.0

string or any reference = null

boolean → false

These values are initialized by the default constructor of JVM during object creation at runtime.

Methods in Java

Method in Java is similar to a function defined in other programming languages. Method describes **behavior of an object**. A method is a collection of statements that are grouped together to perform an operation.

For example, if we have a class Human, then this class should have methods like eating(), walking(), talking() etc, which describes the behavior of the object.

Declaring method is similar to function. See the syntax to declare the method in Java.

```
return-type methodName(parameter-list)

{

    //body of method

}
```

Copy

return-type refers to the type of value returned by the method.

methodName is a valid meaningful name that represent name of a method.

parameter-list represents list of parameters accepted by this method.

Method may have an optional return statement that is used to return value to the caller function.

Example of a Method:

Lets understand the method by simple example that takes a parameter and returns a string value.

```
public String getName(String st)

{

    String name="StudyTonight";

    name=name+st;

    return name;

}
```

Copy

```
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.

Calling a Method

Methods are called to perform the functionality implemented in it. We can call method by its name and store the returned value into a variable.

```
String val = GetName(".com")
```

Copy

It will return a value **studytonight.com** after appending the argument passed during method call.

Returning Multiple values

In Java, we can return multiple values from a method by using array. We store all the values into an array that want to return and then return back it to the caller method. We **must specify return-type as an array** while creating an array. Lets see an example.

Example:

Below is an example in which we return an array that holds multiple values.

```
class MethodDemo2{

    static int[] total(int a, int b)

    {

int[] s = new int[2];

s[0] = a + b;

s[1] = a - b;

        return s;

    }


    public static void main(String[] args)

    {

int[] s = total(200, 70);

System.out.println("Addition = " + s[0]);

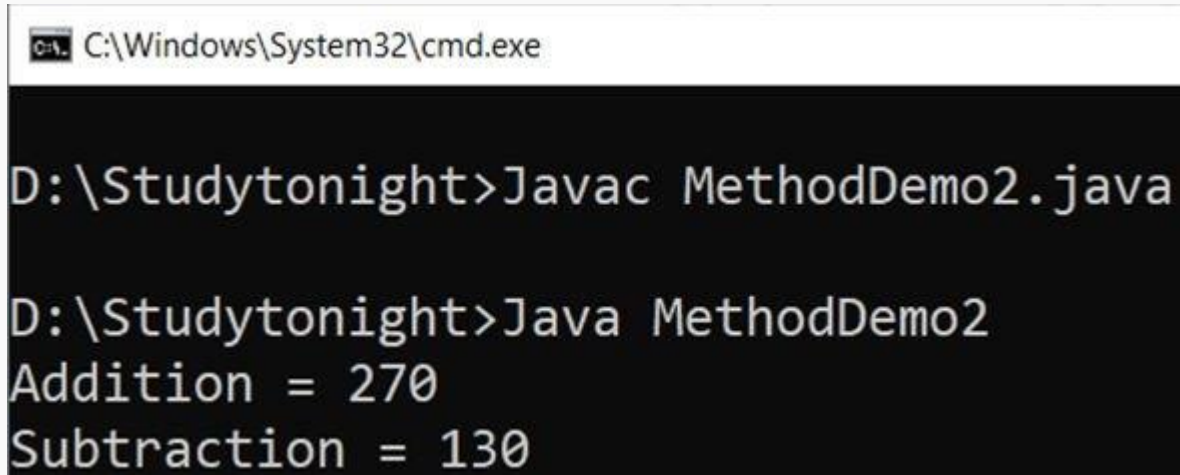
System.out.println("Subtraction = " + s[1]);

    }
```



```
}  
  
}
```

Copy



```
C:\Windows\System32\cmd.exe  
  
D:\Studytonight>Javac MethodDemo2.java  
  
D:\Studytonight>Java MethodDemo2  
Addition = 270  
Subtraction = 130
```

Return Object from Method

In some scenario there can be need to return object of a class to the caller function. In this case, we **must specify class name** in the method definition.

Below is an example in which we are getting an object from the method call. It can also be used to return collection of data.

Example:

In this example, we created a method **get()** that returns object of **Demo** class.

```
class Demo{  
  
    int a;  
  
    double b;  
  
    int c;  
  
    Demo(int m, double d, int a)  
    {  
  
        a = m;  
  
        b = d;  
  
        c = a;  
  
    }  
  
}  
  
class MethodDemo4{
```

```
static Demo get(int x, int y)

{

    return new Demo(x * y, (double)x / y, (x + y));

}

public static void main(String[] args)

{

    Demo ans = get(25, 5);

System.out.println("Multiplication = " + ans.a);

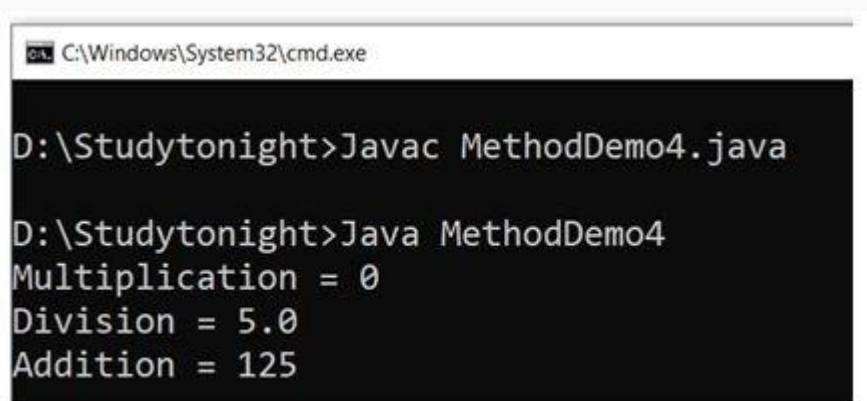
System.out.println("Division = " + ans.b);

System.out.println("Addition = " + ans.c);

    }

}
```

Copy



```
C:\Windows\System32\cmd.exe

D:\Studytonight>Javac MethodDemo4.java

D:\Studytonight>Java MethodDemo4
Multiplication = 0
Division = 5.0
Addition = 125
```

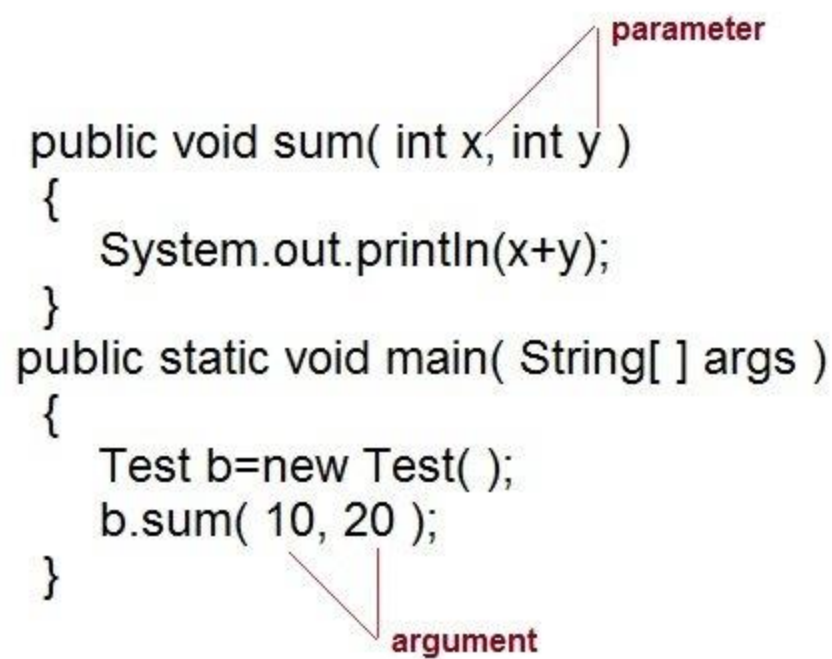
Parameter Vs. Argument in a Method

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.

You can understand it by the below image that explain parameter and argument using a program example.

```
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 );
}
```



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 agrument value.

NOTE :However there is no concept of call-by-reference in Java. Java supports only call by value.

Example of *call-by-value*

Lets see an example in which we are passing argument to a method and modifying its 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
```

See, in the above example, **value passed to the method does not change** even after modified in the method. It shows that **changes made to the value was local** and argument was passed as call-by-value.

Java is Strictly Pass by Value

In Java, it is very confusing whether java is pass by value or pass by reference.

When the values of parameters are copied into another variable, it is known as pass by value and when a reference of a variable is passed to a method then it is known as pass by reference.

It is one of the feature of C language where address of a variable is passed during function call and that reflect changes to original variable.

In case of Java, we there is no concept of variable address, everything is based on object. So either we can pass primitive value or object value during the method call.

Note: Java is strictly pass by value. It means during method call, values are passed not addresses.

Example:

Below is an example in which value is passed by value and changed them inside the function but outside the function changes are not reflected to the original variable values.

```
class Add
{
    int x, y;

    Add(int i, int j)
    {
        x = i;
        y = j;
    }
}

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

```

        Add obj = new Add(5, 10);

        // call by value

        change(obj.x, obj.y);

        System.out.println("x = "+obj.x);

        System.out.println("y = "+obj.y);

    }

    public static void change(int x, int y)

    {

        x++;

        y++;

    }

}

```

Copy

```

x = 5
y = 10

```

Lets take another example in which an object is passed as value to the function, in which we change its variable value and the changes reflected to the original object. It seems like pass by reference but we differentiate it. Here member of an object is changed not the object.

Example:

Below is an example in which value is passed and changes are reflected.

```

class Add

{

    int x, y;

    Add(int i, int j)

    {

        x = i;

        y = j;

    }

}

```

```
}

class Demo

{

    public static void main(String[] args)

    {

        Add obj = new Add(5, 10);

        // call by value (object is passed)

        change(obj);

        System.out.println("x = "+obj.x);

        System.out.println("y = "+obj.y);

    }

    public static void change(Add add)

    {

        add.x++;

        add.y++;

    }

}
```

OUTPUT

```
6
11
```

Constructors in Java

A constructor is a special method that is used to initialize an object. Every class has a constructor either implicitly or explicitly.

If we don't declare a constructor in the class then JVM builds a default constructor for that class. This is known as **default constructor**.

A constructor has **same name as the class name** in which it is declared. **Constructor must have no explicit return type**. Constructor in Java **can not** be **abstract, static, final or synchronized**. These modifiers are not allowed for constructor.

Syntax to declare constructor

```
className (parameter-list) {  
    code-statements  
}
```

Copy

className is the name of class, as **constructor name is same as class name**.

parameter-list is optional, because constructors can be parameterize and non-parameterize as well.

Constructor Example

In Java, constructor structurally looks like given in below program. A Car class has a constructor that provides values to instance variables.

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

Copy

Types of Constructor

Java Supports two types of constructors:

- 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
```

Default Constructor

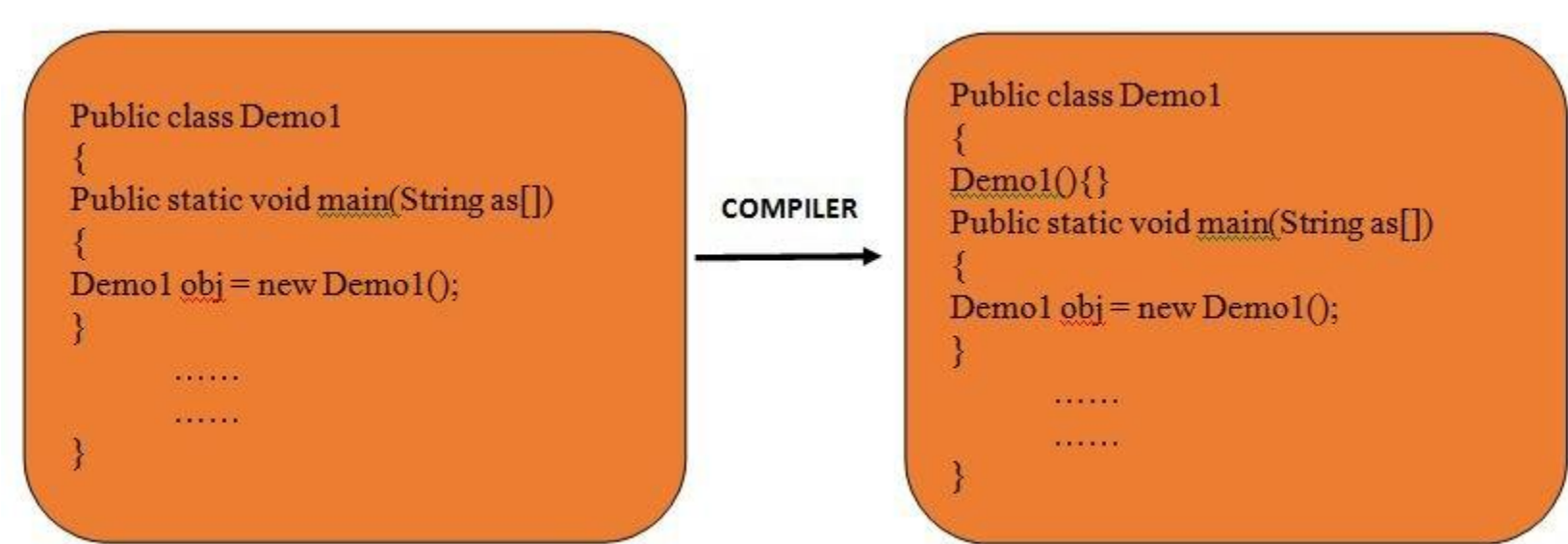
In Java, a constructor is said to be default constructor if it does not have any parameter. Default constructor can be either user defined or provided by JVM.

If a class does not contain any constructor then during runtime JVM generates a default constructor which is known as system define default constructor.

If a class contain a constructor with no parameter then it is known as default constructor defined by user. In this case JVM does not create default constructor.

The **purpose of creating constructor is to initialize states of an object.**

The below image shows how JVM adds a constructor to the class during runtime.



User Define Default Constructor

Constructor which is defined in the class by the programmer is known as user-defined default constructor.

Example:

In this example, we are creating a constructor that has same name as the class name.

```
class AddDemo1
{
    AddDemo1 ()
    {
        int a=10;
        int b=5;
        int c;
        c=a+b;
    }
}
```



```

        System.out.println("*****Default Constructor*****");

        System.out.println("Total of 10 + 5 = "+c);

    }

    public static void main(String args[])

    {

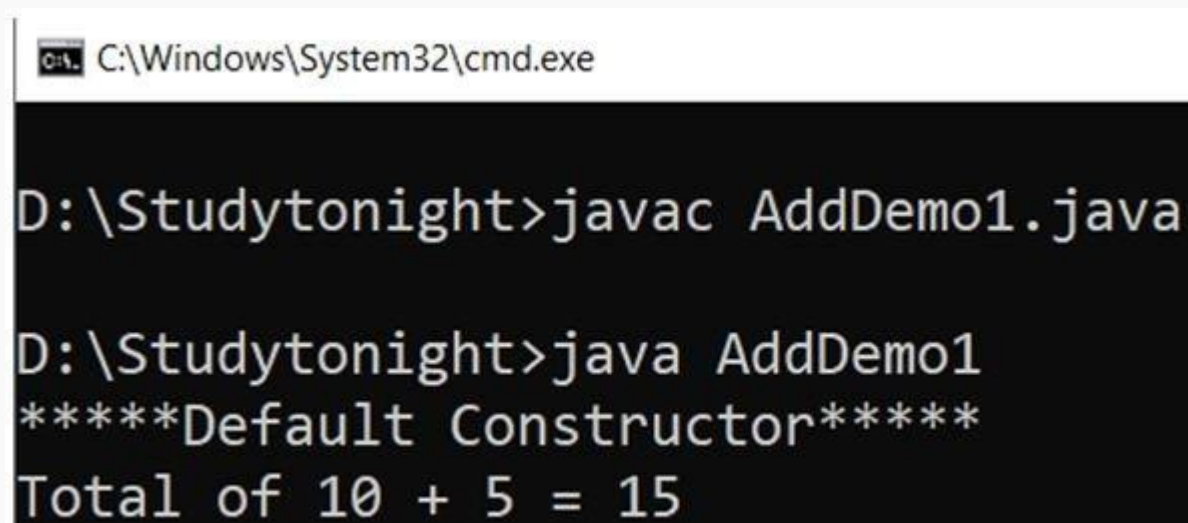
        AddDemo1 obj=new AddDemo1();

    }

}

```

Copy



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac AddDemo1.java

D:\Studytonight>java AddDemo1
*****Default Constructor*****
Total of 10 + 5 = 15

```

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**.

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);

}

}
```

Copy

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

Constructor Chaining

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

To call constructor from another constructor **this keyword** is used. This keyword is used to refer current object.

Lets see an example to understand constructor chaining.

```
class Test

{

    Test()

    {

        this(10);

    }

    Test(int x)

    {

        System.out.println("x="+x);

    }

}
```

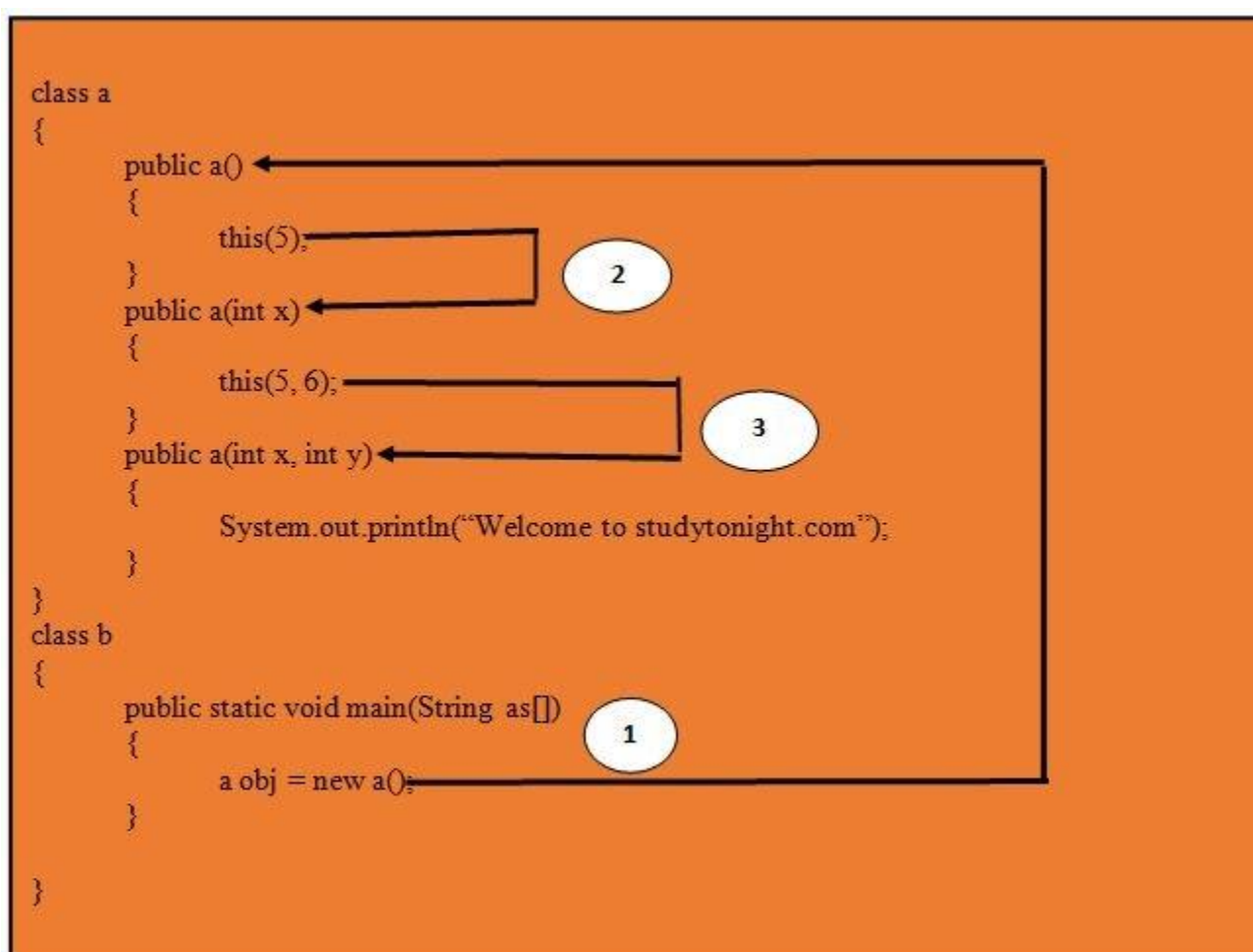
```
public static void main(String arg[])
{
Test object = new Test();
}
}
```

Copy

```
x=10
```

Constructor chaining is used when we want to perform multiple tasks by creating a single object of the class.

In the below image, we have described the flow of constructor calling in the same class.



Example:

Lets see one more example to understand the constructor chaining. Here we have created three constructors and calling them using by using this keyword.

```
class abc
{
    public abc()
    {
        this(5);
    }
}
```

```

        System.out.println("Default Constructor");

    }

    public abc(int x)

    {

        this(5, 6);

        System.out.println("Constructor with one Parameter");

        System.out.println("Value of x ==> "+x);

    }

    public abc(int x, int y)

    {

        System.out.println("Constructor with two Parameter");

        System.out.println("Value of x and y ==> "+x+" "+y);

    }

}

class ChainingDemo1

{

    public static void main(String as[])

    {

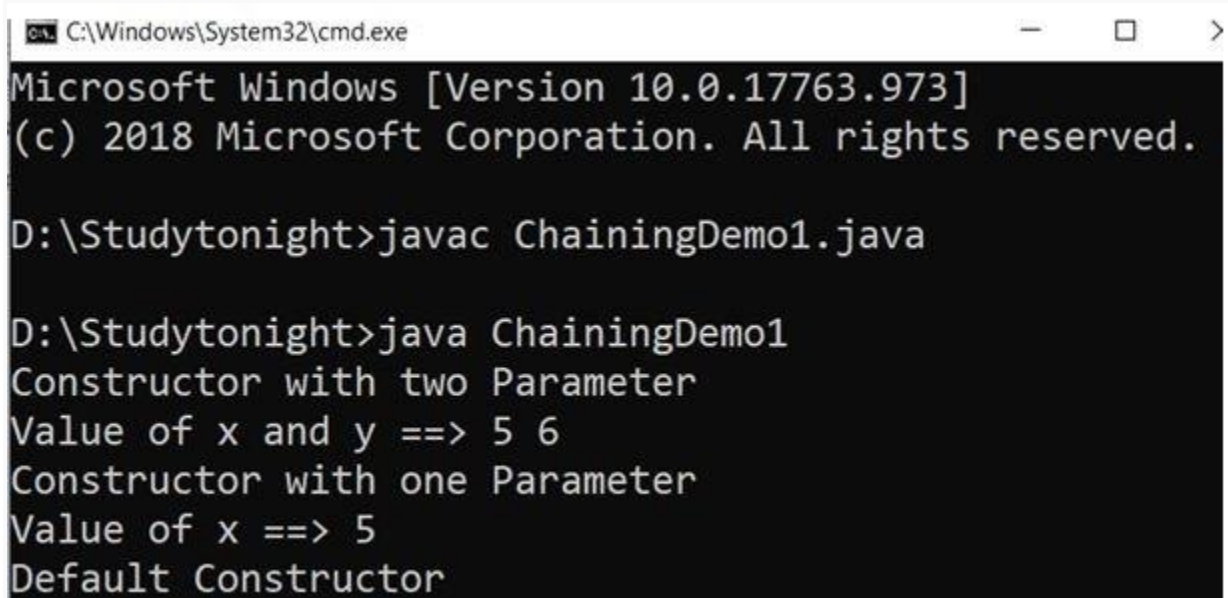
        abcobj = new abc();

    }

}

```

Copy



```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

D:\Studytonight>javac ChainingDemo1.java

D:\Studytonight>java ChainingDemo1
Constructor with two Parameter
Value of x and y ==> 5 6
Constructor with one Parameter
Value of x ==> 5
Default Constructor

```

Private Constructors

In Java, we can create private constructor to **prevent class being instantiate**. It means by declaring a private constructor, it restricts to create object of that class.

Private constructors are used to create **singleton class**. A class which can have only single object known as singleton class.

In private constructor, only one object can be created and the object is created within the class and also all the methods are static. An object can not be created if a private constructor is present inside a class. A class which have a private constructor and all the methods are static then it is called **Utility class**.

Example:

```
final class abc
{
    private abc()
    {}

    public static void add(int a, int b)
    {
        int z = a+b;

        System.out.println("Addition: "+z);
    }

    public static void sub(int x, int y)
    {
        int z = x-y;

        System.out.println("Subtraction: "+z);
    }

}

class PrivateConDemo
{
    public static void main(String as[])
    {
        abc.add(4, 5);
    }
}
```

```
    abc.sub(5, 3);  
  
}  
  
}
```

Copy



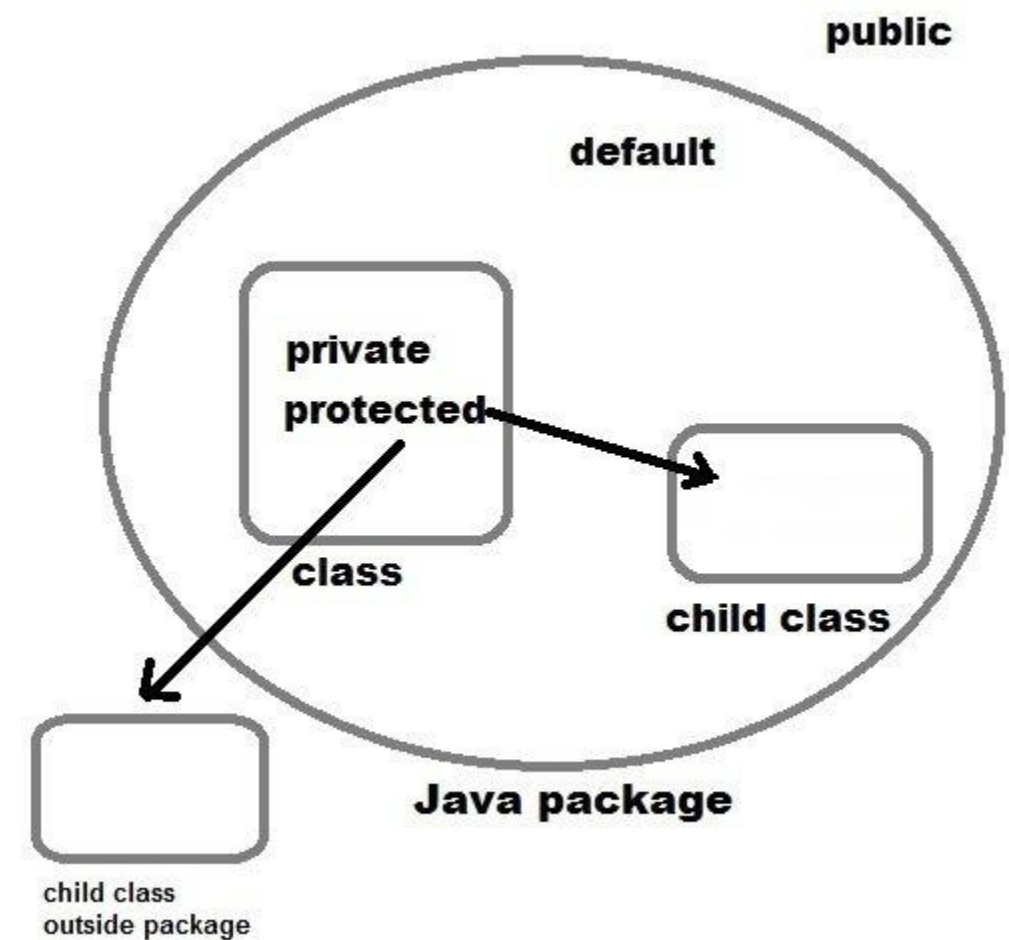
Access Modifiers in Java

Access modifiers are keywords in Java that are used to set accessibility. An access modifier restricts the access of a class, constructor, data member and method in another class.

Java language has four access modifier to control access level for classes and its members.

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

Java also supports many non-access modifiers, such as **static**, **abstract**, **synchronized**, **native**, **volatile**, **transient** etc. We will cover these in our other tutorial.



Default Access Modifier

If we don't specify any access modifier then it is **treated as default** modifier. It is used to set accessibility within the package. It means we can not access its method or class from outside the package. It is also known as package accessibility modifier.

Example:

In this example, we created a **Demo class** inside the **package1** and another class **Test** by which we are accessing show() method of Demo class. We did not mention access modifier for the show() method that's why it is not accessible and reports an error during compile time.

Demo.java

```
package package1;

public class Demo {

    int a = 10;

    // default access modifier

    void show() {

        System.out.println(a);

    }

}
```

Copy

Test.java

```
import package1.Demo;

public class Test {

    public static void main(String[] args) {

        Demo demo = new Demo();

        demo.show(); // compile error

    }

}
```



```
}
```

Copy

The method show() from the type Demo is not visible

Public Access Modifier

public access modifier is used to set public accessibility to a variable, method or a class. Any variable or method which is declared as public can be accessible from anywhere in the application.

Example:

Here, we have two class Demo and Test located in two different package. Now we want to access show method of Demo class from Test class. The method has public accessibility so it works fine. See the below example.

Demo.java

```
package package1;

public class Demo {

    int a = 10;

    // public access modifier

    public void show() {

        System.out.println(a);

    }

}
```

Copy

Test.java

```
package package2;

import package1.Demo;

public class Test {
```

```
public static void main(String[] args) {  
  
    Demo demo = new Demo();  
  
    demo.show();  
  
}  
  
}
```

Copy

10

Protected Access Modifier

Protected modifier protects the variable, method from accessible from outside the class. It is accessible within class, and in the child class (inheritance) whether child is **located** in the **same package** or some **other package**.

Example:

In this example, Test class is extended by Demo and called a protected method show() which is accessible now due to inheritance.

Demo.java

```
package package1;  
  
public class Demo {  
  
    int a = 10;  
  
    // public access modifier  
    protected void show() {  
  
        System.out.println(a);  
  
    }  
  
}
```

Copy

Test.java

```
package package2;

import package1.Demo;

public class Test extends Demo{

    public static void main(String[] args) {

        Test test = new Test();

        test.show();

    }

}
```

Copy

10

Private Access Modifier

Private modifier is most restricted modifier which allows accessibility within same class only. We can set this modifier to any variable, method or even constructor as well.

Example:

In this example, we set private modifier to show() method and try to access that method from outside the class. Java does not allow to access it from outside the class.

Demo.java

```
class Demo {

    int a = 10;

    private void show() {

        System.out.println(a);

    }

}
```

Copy

Test.java

```
public class Test {

    public static void main(String[] args) {

        Demo demo = new Demo();

        demo.show(); // compile error

    }

}
```

Copy

The method show() from the type Demo is not visible

Non-access Modifier

Along with access modifiers, Java provides non-access modifiers as well. These modifier are **used to set special properties** to the variable or method.

Non-access modifiers do not change the accessibility of variable or method, but they provide special properties to them. Java provides following non-access modifiers.

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

Final Modifier

Final modifier can be used with **variable, method** and class. if **variable** is declared **final** then we **cannot change** its value. If **method** is declared **final** then it **can not be overridden** and if a **class** is declared **final** then we **can not inherit it**.

Static modifier

static modifier is used to make field static. We can use it to declare static variable, method, class etc. static can be use to declare class level variable. If a method is declared static then we don't need to have object to access that. We can use static to create nested class.

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.

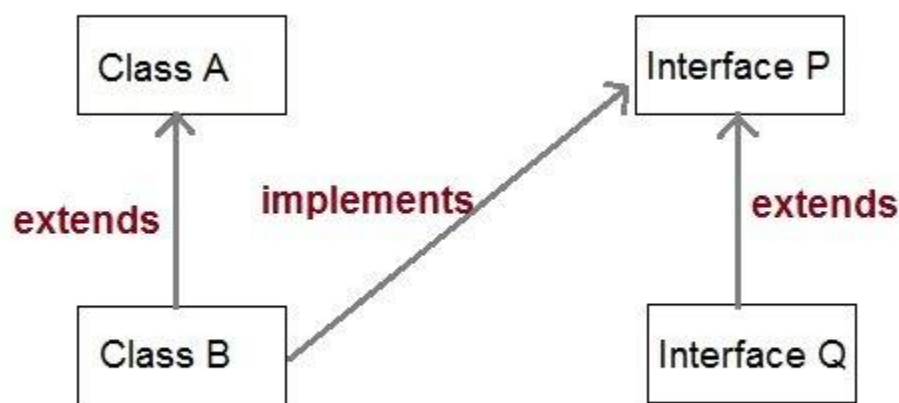
Volatile modifier

Volatile modifier tells to the compiler that the volatile variable can be changed unexpectedly by other parts of a program. Volatile variables are used in case of multi-threading program. **volatile** keyword cannot be used with a method or a class. It can be only used with a variable.

Inheritance (IS-A relationship) in Java

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 **extends** keyword is used to achieve Inheritance. It shows super class and sub-class relationship.

```
class Vehicle
{
    .....
}

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

Copy

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. It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
 2. It promotes polymorphism by allowing method overriding.
-

Disadvantages of Inheritance

Main disadvantage of using inheritance is that the two classes (parent and child class) gets **tightly coupled**.

This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, **it cannot be independent of each other**.

Simple example of Inheritance

Before moving ahead let's take a quick example and try to understand the concept of Inheritance better,

```
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

}

}

```

Copy

Child method

Parent method

In the code above we have a class **Parent** which has a method **p1()**. We then create a new class **Child** which inherits the class **Parent** using the **extends** keyword and defines its own method **c1()**. Now by virtue of inheritance the class **Child** can also access the **public** method **p1()** of the class **Parent**.

Inheriting variables of super class

All the members of super class implicitly inherits to the child class. Member consists of instance variable and methods of the class.

Example

In this example the sub-class will be accessing the variable defined in the super class.

```

class Vehicle

{

    // variable defined

    String vehicleType;

}

public class Car extends Vehicle {

    String modelType;

    public void showDetail()

    {

        vehicleType = "Car";    //accessing Vehicle class member
variable

        modelType = "Sports";

        System.out.println(modelType + " " + vehicleType);

    }

}

```

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

{

    Car car = new Car();

    car.showDetail();

}

}
```

Copy

Sports Car

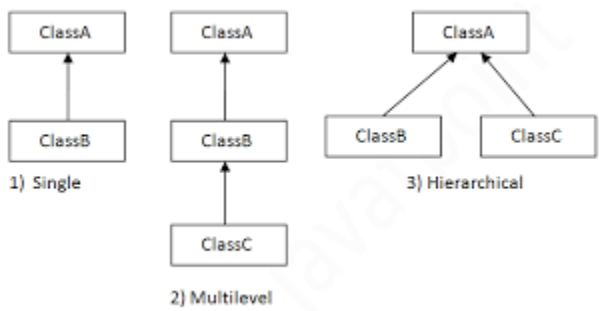
Types of Inheritance

Java mainly supports only three types of inheritance that are listed below.

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

NOTE: Multiple inheritance is not supported in java

We can get a quick view of type of inheritance from the below image.



Single Inheritance

When a class extends to another class then it forms single inheritance. In the below example, we have two classes in which class A extends to class B that forms single inheritance.

```
class A{

    int a = 10;

    void show() {

        System.out.println("a = "+a);

    }

}
```



```
public class B extends A{

    public static void main(String[] args) {

        B b = new B();

        b.show();

    }

}
```

Copy

```
a=10
```

Here, we can notice that show() method is declared in class A, but using child class Demo object, we can call it. That shows the inheritance between these two classes.

Multilevel Inheritance

When a class extends to another class that also extends some other class forms a multilevel inheritance. For example a class C extends to class B that also extends to class A and all the data members and methods of class A and B are now accessible in class C.

Example:

```
class A{

    int a = 10;

    void show() {

        System.out.println("a = "+a);

    }

}

class B extends A{

    int b = 10;

    void showB() {

        System.out.println("b = "+b);

    }

}
```

```

    }

}

public class C extends B{

    public static void main(String[] args) {

        C c = new C();

        c.show();

        c.showB();

    }

}

```

Copy

```

a=10

b=10

```

Hierarchical Inheritance

When a class is extended by two or more classes, it forms hierarchical inheritance. For example, class B extends to class A and class C also extends to class A in that case both B and C share properties of class A.

```

class A{

    int a = 10;

    void show() {

        System.out.println("a = "+a);

    }

}

class B extends A{

    int b = 10;

    void showB() {

        System.out.println("b = "+b);

    }

}

```

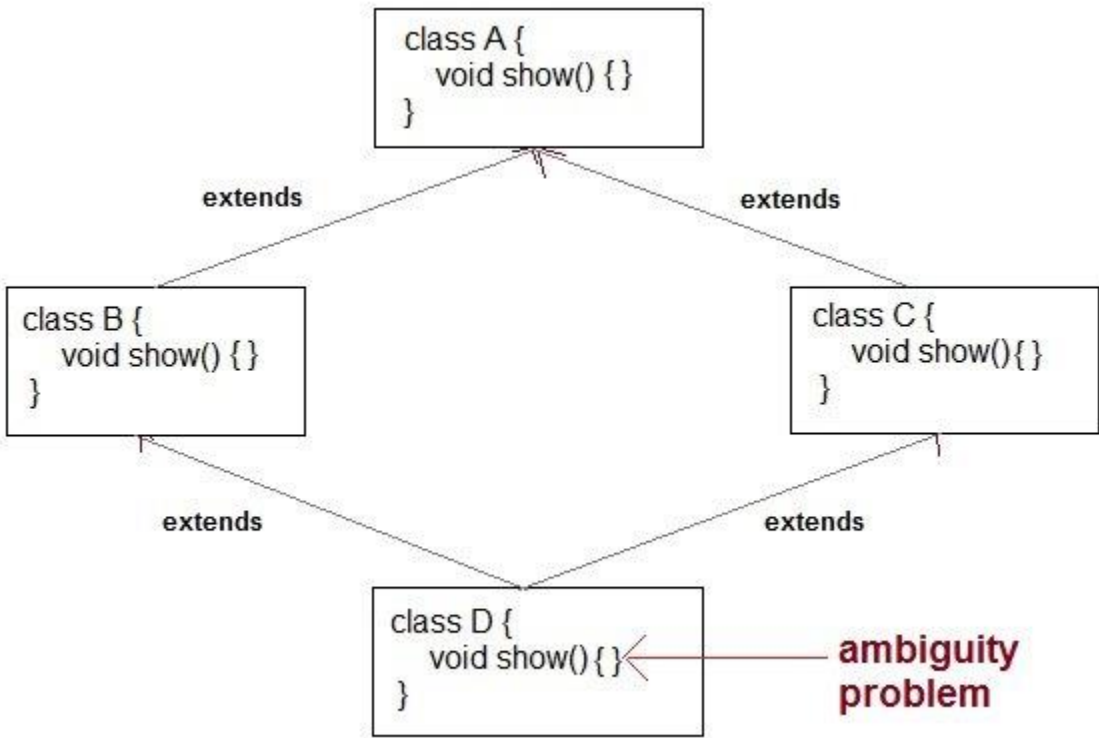
```
    }  
  
}  
  
public class C extends A{  
  
    public static void main(String[] args) {  
  
        C c = new C();  
  
        c.show();  
  
        B b = new B();  
  
        b.show();  
  
    }  
  
}
```

Copy

```
a = 10  
a = 10
```

Why multiple inheritance is not supported in Java?

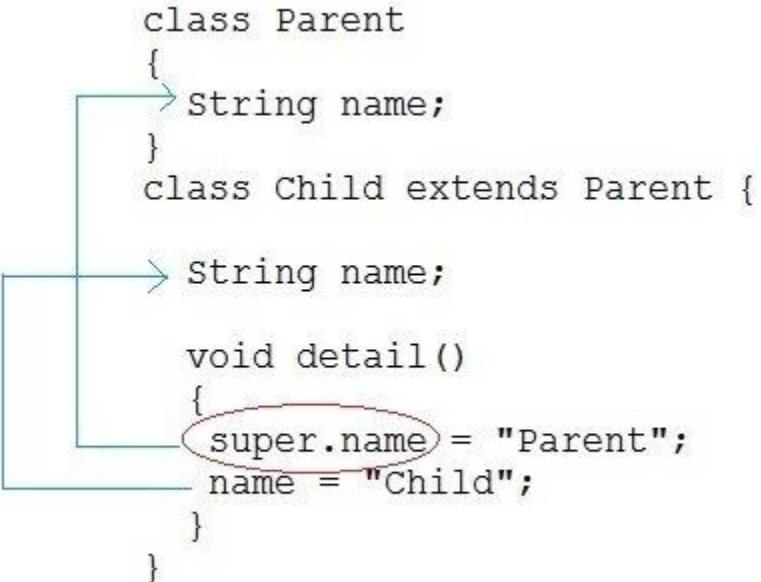
- To remove ambiguity.
- To provide more maintainable and clear design.



super keyword

In Java, **super** keyword is used to refer to immediate parent class of a child 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**

In this examble we will only focus on accessing the parent class property or variables.

```
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();
    }
}
```

Copy

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

In this examle we will only focus on accessing the parent class methods.

```
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();
    }
}
```

Copy

Parent

Child

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

In this examle we will focus on accessing the parent class constructor.

```
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();
    }
}
```

```
}
```

Copy

Parent and Child

Note: When calling the parent class constructor from the child class using super keyword, super keyword should always be the first line in the method/constructor of the child class.

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 relationship) in Java

Aggregation is a term which is used to refer **one way relationship** between two objects. For example, **Student** class can have **reference** of **Address** class but vice versa does not make sense.

In Java, aggregation represents **HAS-A relationship**, which means when a class contains reference of another class known to have aggregation.

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

Lets understand it by an example and consider two classes Student and Address. Each student has own address that makes has-a relationship but address has student not makes any sense. We can understand it more clearly using Java code.

```
Class Address{

    int street_no;

    String city;

    String state;

    int pin;

    Address(int street_no, String city, String state, int pin ){

        this.street_no = street_no;

        this.city = city;

        this.state = state;

        this.pin = pin;

    }

}

class Student

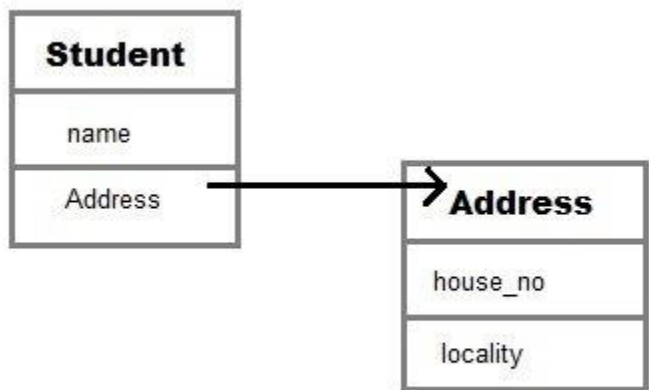
{

    String name;
```

```
Address ad;  
  
}
```

Copy

Here in the above code, we can see **Student** class has-a relationship with **Address** class. We have draw an image too to demonstrate relation between these two classes..



The **Student** class has an instance variable of type **Address**. As we have a variable of type **Address** in the **Student** class, it can use Address reference which is **ad** in this case, to invoke methods of the **Address** class.

Advantage of Aggregation

The main advantage of using aggregation is to maintain **code re-usability**. If an entity has a relation with some other entity than it can reuse code just by referring that.

Aggregation Example

Now lets understand it by using an example, here we created two classes Author and Book and Book class has a relation with Author class by having its reference. Now we are able to get all the properties of both class.

```
class Author  
{  
  
    String authorName;  
  
    int age;  
  
    String place;  
  
    // Author class constructor  
    Author(String name, int age, String place)  
  
    {  
  
        this.authorName = name;  
  
        this.age = age;  
  
    }  
}
```



```
        this.place = place;

    }

}

class Book

{

    String name;

    int price;

    // author details

    Author auther;

    Book(String n, int p, Author auther)

    {

        this.name = n;

        this.price = p;

        this.auther = auther;

    }

    public static void main(String[] args) {

        Author auther = new Author("John", 42, "USA");

        Book b = new Book("Java for Begginer", 800, auther);

        System.out.println("Book Name: "+b.name);

        System.out.println("Book Price: "+b.price);

        System.out.println("-----Auther Details-----");

        System.out.println("Auther Name: "+b.auther.authorName);

        System.out.println("Auther Age: "+b.auther.age);

        System.out.println("Auther place: "+b.auther.place);

    }

}
```

Copy

Book Name: Java for Begginer

Book Price: 800

-----Author Details-----

Auther Name: John

Auther Age: 42

Auther place: USA

Lets take one more example to understand aggregation. Suppose we have one more class Publisher then the Book class can reuse Publisher class details by just using its reference as Author class. Lets understand it by Java code.

```
class Publisher{

    String name;

    String publisherID;

    String city;

    Publisher(String name, String publisherID, String city) {

        this.name = name;

        this.publisherID = publisherID;

        this.city = city;

    }

}

class Author

{

    String authorName;

    int age;

    String place;

    // Author class constructor

    Author(String name, int age, String place)

    {

        this.authorName = name;
```

```

        this.age = age;

        this.place = place;
    }
}

class Book
{
    String name;

    int price;

    // author details

    Author auther;

    Publisher publisher;

    Book(String n, int p, Author auther, Publisher publisher )
    {
        this.name = n;

        this.price = p;

        this.auther = auther;

        this.publisher = publisher;
    }

    public static void main(String[] args) {

        Author auther = new Author("John", 42, "USA");

        Publisher publisher = new Publisher("Sun Publication", "JDSR-III4",
"LA");

        Book b = new Book("Java for Begginer", 800, auther, publisher);

        System.out.println("Book Name: "+b.name);

        System.out.println("Book Price: "+b.price);

        System.out.println("-----Author Details-----");

        System.out.println("Auther Name: "+b.auther.authorName);

        System.out.println("Auther Age: "+b.auther.age);

        System.out.println("Auther place: "+b.auther.place);

        System.out.println("-----Publisher Details-----");
    }
}

```

```
        System.out.println("Publisher Name: "+b.publisher.name);

        System.out.println("Publisher ID: "+b.publisher.publisherID);

        System.out.println("Publisher City: "+b.publisher.city);

    }

}
```

Copy

```
Book Name: Java for Begginer
Book Price: 800
-----Author Details-----
Auther Name: John
Auther Age: 42
Auther place: USA
-----Publisher Details-----
Publisher Name: Sun Publication
Publisher ID: JDSR-III4
Publisher City: LA
```

Composition in Java

Composition is a more restricted form of Aggregation. Composition can be described as when one class which includes another class, is dependent on it in such a way that it cannot functionally exist without the class which is included. For example a class **Car** cannot exist without **Engine**, as it won't be functional anymore.

Hence the word **Composition** which means the items something is made of and if we change the composition of things they change, similarly in Java classes, one class including another class is called a composition if the class included provides core functional meaning to the outer class.

```
class Car

{

    private Engine engine;

    Car(Engine en)

    {

        engine = en;

    }

}
```

Copy

Here by examining code, we can understand that if Car class does not have relationship with Engine class then Car does not have existence.

Composition is a **design technique**, not a feature of Java but we can achieve it using Java code.

Q. When to use Inheritance and Aggregation?

When you want to use some property or behaviour of any class without the requirement of modifying it or adding more functionality to it, in such case **Aggregation** is a better option because in case of Aggregation we are just using any external class inside our class as a variable.

Whereas when you want to use and modify some property or behaviour of any external class or may be want to add more function on top of it, its best to use **Inheritance**.

To understand more on inheritance, you can visit our detailed tutorial here. [Click Here to see Inheritance in Java](#)

Method Overloading in Java

Method overloading **is a concept** that allows to declare **multiple methods with same name but different parameters in the same class**.

Java supports method overloading and always occur in the same class(unlike method overriding).

Method overloading is one of the ways through which java supports polymorphism. Polymorphism is a concept of object oriented programming that deal with multiple forms. We will cover polymorphism in separate topics later on.

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** can not be overloaded.

Note: Overloaded method can have different access modifiers and it does not have any significance in method overloading.

There are two different ways of method overloading.

1. Different datatype of arguments
2. Different number of arguments

Method overloading by changing data type of arguments.

Example:

In this example, we have two sum() methods that take integer and float type arguments respectively.

Notice that in the same class we have two **methods with same name but different types of parameters**

```
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 (4.6f, 3.8f); //sum(float a, float b) is called.

}

}

```

Copy

```

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:

In this example, we have two methods

```

class Demo

{

    void multiply(int l, int b)

    {

        System.out.println("Result is" + (l*b)) ;

    }

    void multiply(int l, int b, int h)

```

```

{

    System.out.println("Result is" + (l * b * h));

}

public static void main(String[] args)

{

    Demo ar = new Demo();

    ar.multiply(8, 5);    //multiply(int l, int b) is method is called

    ar.multiply(4, 6, 2);    //multiply(int l, int b, int h) is called

}

}

```

Copy

```
Result is 40
```

```
Result is 48
```

In this example the `multiply()` method is overloaded twice. The first method takes two arguments and the second method takes three arguments.

When an overloaded method is called Java look for match between the arguments to call the method and the its 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.

Java supports automatic type promotion, like int to long or float to double etc. In this example we are doing same and calling a function that takes **one integer** and **second long type** argument. At the time of calling we passed integer values and Java treated second argument as long type. See the below example.

```

class Demo

{

    void sum(int l, long b)

    {

        System.out.println("Sum is" + (l + b)) ;

    }

    void sum(int l, int b, int h)

    {

        System.out.println("Sum is" + (l + b + h)) ;

    }

}

```

```
}

public static void main (String[] args)

{

    Demo  ar = new Demo();

    ar.sum(8,5);

}

}
```

Copy

Sum is 13

Method overloading if the order of parameters is changed

We can have two methods with same name and parameters but the order of parameters is different.

Example:

In this scenario, method overloading works but internally JVM treat it as method having different type of arguments

```
class Demo{

    public void get(String name,  int id)

    {

System.out.println("Company Name :"+ name);

        System.out.println("Company Id :"+ id);

    }

    public void get(int id, String name)

    {

System.out.println("Company Id :"+ id);

        System.out.println("Company Name :"+ name);

    }

}
```



```
class MethodDemo8{

    public static void main (String[] args) {

        Demo obj = new Demo();

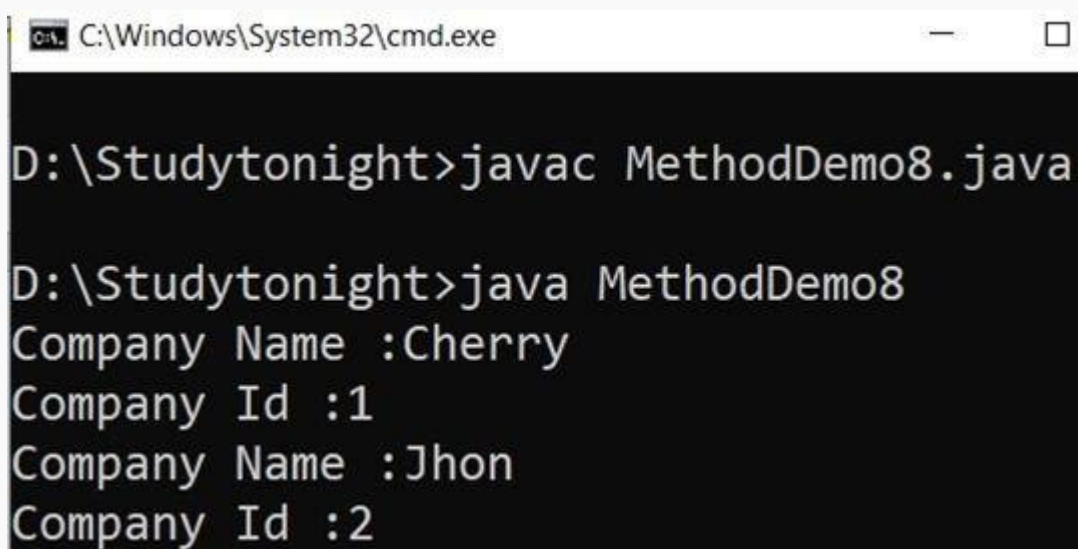
        obj.get("Cherry", 1);

        obj.get("Jhon", 2);

    }

}
```

Copy



```
C:\Windows\System32\cmd.exe

D:\Studytonight>javac MethodDemo8.java

D:\Studytonight>java MethodDemo8
Company Name :Cherry
Company Id :1
Company Name :Jhon
Company Id :2
```

Overloading main Method

In Java, we can overload the main() method using different number and types of parameter but the JVM only understand the original main() method.

Example:

In this example, we created three main() methods having different parameter types.

```
public class MethodDemo10{

    public static void main(intargs)

    {

        System.out.println("Main Method with int argument Executing");

        System.out.println(args);

    }

    public static void main(char args)

    {

        System.out.println("Main Method with char argument Executing");

    }

}
```

```

    System.out.println(args);

}

    public static void main(Double[] args)

    {
System.out.println("Main Method with Double Executing");

    System.out.println(args);

    }

    public static void main(String[] args)

    {
System.out.println("Original main Executing");

    MethodDemo10.main(12);

    MethodDemo10.main('c');

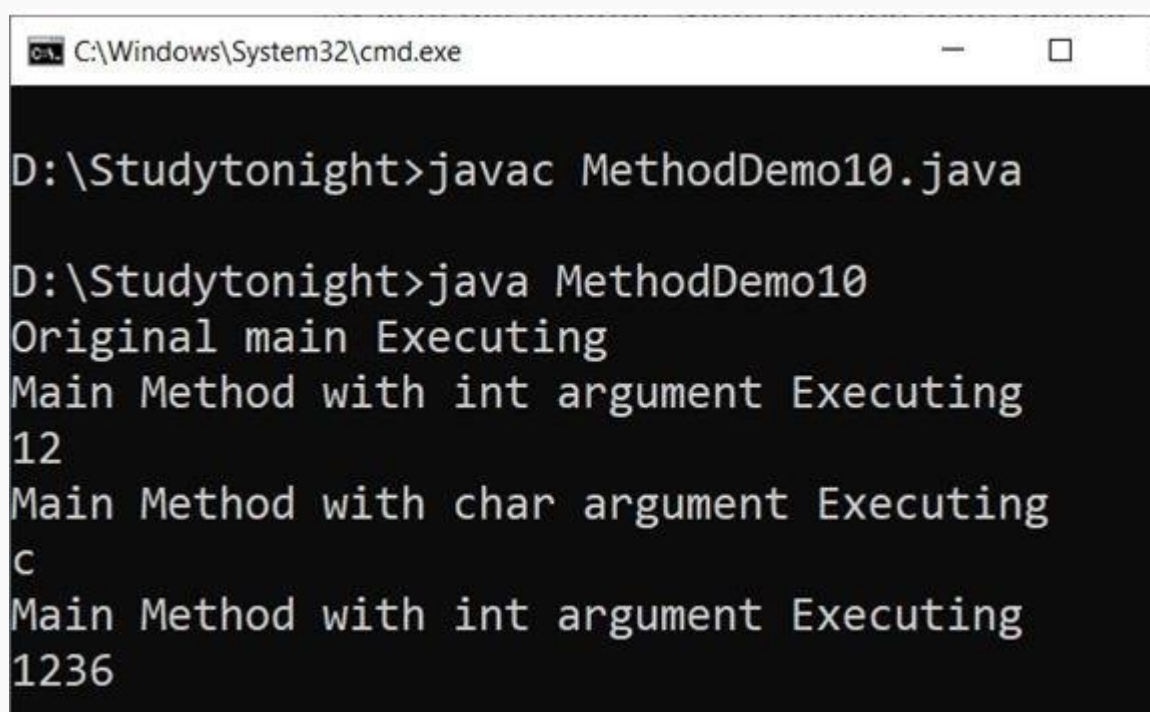
    MethodDemo10.main(1236);

    }

}

```

Copy



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac MethodDemo10.java

D:\Studytonight>java MethodDemo10
Original main Executing
Main Method with int argument Executing
12
Main Method with char argument Executing
c
Main Method with int argument Executing
1236

```

Method overloading and null error

This is a general issue when working with objects, if same name methods having reference type parameters then be careful while passing arguments.

Below is an example in which you will know how a null value can cause an error when methods are overloaded.

Example:

Null value is a default value for all the reference types. It created ambiguity to JVM that reports error.

```
public class Demo
{
    public void test(Integer i)
    {
        System.out.println("test(Integer ) ");
    }

    public void test(String name)
    {
        System.out.println("test(String ) ");
    }

    public static void main(String [] args)
    {
        Demo obj = new Demo();

        obj.test(null);
    }
}
```

Copy

The method test(Integer) is ambiguous for the type Demo

Reason:

The main reason for getting the compile time error in the above example is that here we have Integer and String as arguments which are not primitive data types in java and this type of argument do not accept any null value. When the null value is passed the compiler gets confused which method is to be selected as both the methods in the above example are accepting null.

However, we can **solve this to pass specific reference type rather than value.**

Example:

In this example, we are passing specific type argument rather than null value.

```
public class MethodDemo9
{
    public void test(Integer i)
    {
        System.out.println("Method ==> test(Integer)");
    }

    public void test(String name)
    {
        System.out.println("Method ==> test(String) ");
    }

    public static void main(String [] args)
    {
        MethodDemo9 obj = new MethodDemo9 ();

        Integer a = null;
        obj.test(a);

        String b = null;
        obj.test(b);
    }
}
```

Copy

A screenshot of a Windows command prompt window. The title bar shows 'C:\Windows\System32\cmd.exe'. The command prompt shows the following sequence of commands and output:
D:\Studytonight>javac MethodDemo9.java
D:\Studytonight>java MethodDemo9
Method ==> test(Integer)
Method ==> test(String)

```
C:\Windows\System32\cmd.exe
D:\Studytonight>javac MethodDemo9.java
D:\Studytonight>java MethodDemo9
Method ==> test(Integer)
Method ==> test(String)
```

Method Overriding in Java

Method overriding is a process of overriding base class method by derived class method with more specific definition.

Method overriding performs only if two classes have **is-a** relationship. It means class must have inheritance. In other words, It is performed between two classes using inheritance relation.

In overriding, method of both class **must** have **same name** and equal number of **parameters**.

Method overriding is also referred to as **runtime polymorphism** because calling method is decided by JVM during runtime.

The key benefit of overriding is the ability to **define method that's specific to a particular subclass type**.

Rules for Method Overriding

1. Method name must be same for both parent and child classes.
2. Access modifier of child method must not be restrictive than parent class method.
3. Private, final and static methods cannot be overridden.
4. There must be an IS-A relationship between classes (inheritance).

Example of Method Overriding

Below we have simple code example with one parent class and one child class wherein the child class will override the method provided by the parent class.

```
class Animal
{
    public void eat()
    {
        System.out.println("Eat all eatables");
    }
}

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

    public static void main(String[] args) {
        Dog d = new Dog();
    }
}
```

```
        d.eat();
    }
}
```

Copy

```
Dog like to eat meat
```

As you can see here **Dog** class gives it own implementation of `eat()` method. For method overriding, the method must have same name and same type signature in both parent and child class.

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

Example: Access modifier is more restrictive in child class

Java does not allows method overriding if child class has more restricted access modifier than parent class.

In the below example, to the child class method, we set protected which is restricted than public specified in parent class.

```
class Animal
{
    public void eat()
    {
        System.out.println("Eat all eatables");
    }
}

class Dog extends Animal
{
    protected void eat() //error
    {
        System.out.println("Dog like to eat meat");
    }

    public static void main(String[] args) {
        Dog d = new Dog();
    }
}
```

```
        d.eat();
    }
}
```

Copy

Cannot reduce the visibility of the inherited method from Animal.

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 getObj()
    {
        System.out.println("Animal object");
        return new Animal();
    }
}

class Dog extends Animal
{
    Dog getObj() //Legal override after Java5 onward
    {
        System.out.println("Dog object");
        return new Dog();
    }

    public static void main(String[] args) {
        new Dog().getObj();
    }
}
```

```
    }  
  
}
```

Copy

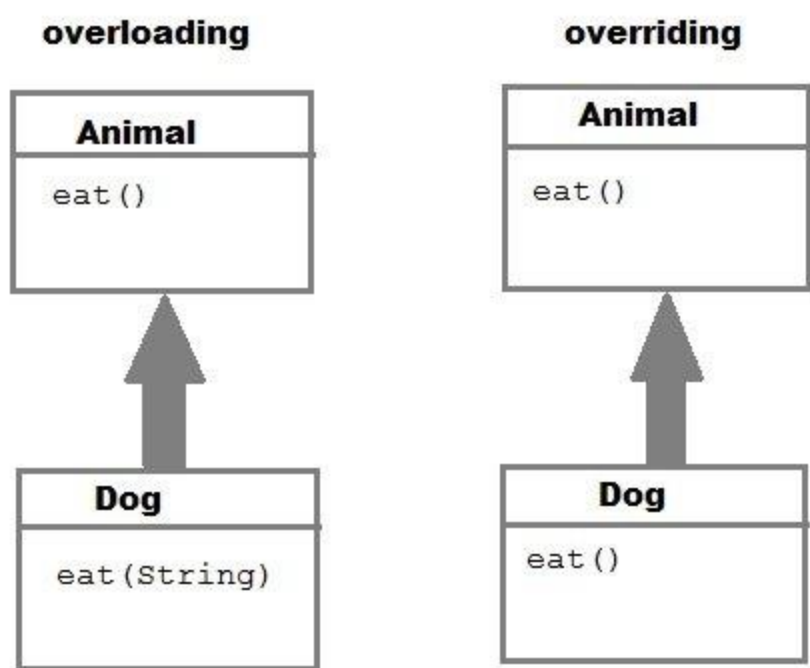
Dog object

Difference between Overloading and Overriding

Method overloading and Method overriding seems to be similar concepts but they are not. Let's see some differences between both of them:

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 cannot be more restrictive than original method(can be less restrictive).
It is Compiled Time Polymorphism.	It is Run Time Polymorphism.
It is performed within a class	It is performed between two classes using inheritance
It is performed between two classes using inheritance relation.	It requires always inheritance.
It should have methods with the same name but a different signature.	It should have methods with same name and signature
It can not have the same return type.	It should always have the same return type.
It can be performed using the static method	It can not be performed using the static method

Method Overloading	Method Overriding
It uses static binding	It uses the dynamic binding.
Access modifiers and Non-access modifiers can be changed.	Access modifiers and Non-access modifiers can not be changed.
It is code refinement technique.	It is a code replacement technique.
No keywords are used while defining the method.	Virtual keyword is used in the base class and override keyword is used in the derived class.
Private, static, final methods can be overloaded	Private, static, final methods can not be overloaded
No restriction is Throws Clause.	Restriction in only checked exception.
It is also known as Compile time polymorphism or static polymorphism or early binding	It is also known as Run time polymorphism or Dynamic polymorphism or Late binding
Example	Example: <pre>class Demo2{ void a() {System.out.println("A");}}</pre> <pre>class b extends c {void a(){System.out.println("B");}}</pre>
Copy	Copy



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.

Example: Overriding toString()

The toString() method of Object class is used to return string representation of an object.

Since object is super class of all java classes then we can override its string method to provide our own string presentation.

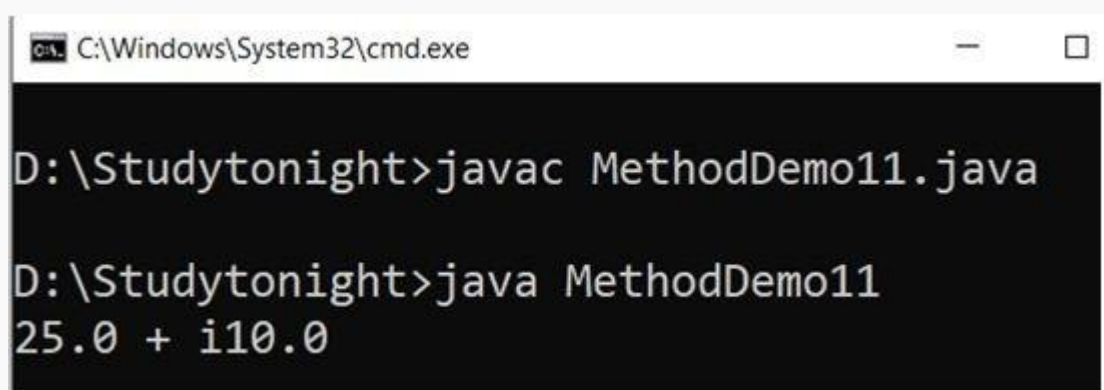
If we don't override string class and print object reference then it prints some hash code in "class_name @ hash code" format.

Below is an example of the overriding toString() method of **Object** class.

```
class Demo{  
    private double a, b;  
    public Demo(double a, double b)  
    {  
        this.a = a;  
        this.b = b;  
    }  
    @Override  
    public String toString()  
}
```

```
{  
  
    return String.format(a + " + i" + b);  
  
}  
  
}  
  
public class MethodDemo11{  
  
    public static void main(String[] args)  
  
    {  
  
        Demo obj1 = new Demo(25, 10);  
System.out.println(obj1);  
  
    }  
  
}
```

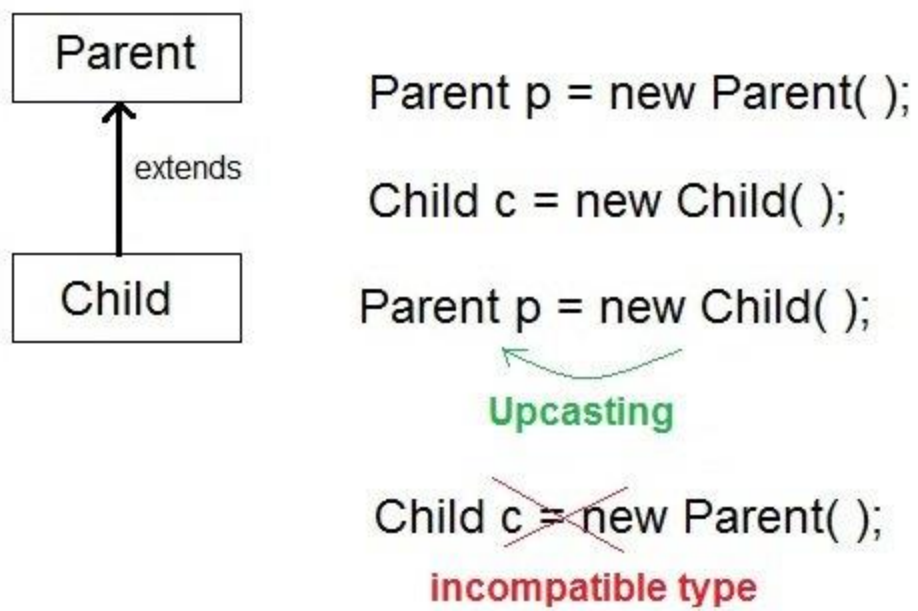
Copy



```
C:\Windows\System32\cmd.exe  
  
D:\Studytonight>javac MethodDemo11.java  
  
D:\Studytonight>java MethodDemo11  
25.0 + i10.0
```

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.



Upcasting in Java

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**. In Java this can be done and is helpful in scenarios where multiple child classes extends one parent class. In those cases we can create a parent class reference and assign child class objects to it.

Let's take an example to understand it,

```
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

    }

}

```

Copy

Indoor & outdoor

Outdoor game

Outdoor game

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

In simpler terms, Static binding means when the type of object which is invoking the method is determined at compile time by the compiler. While Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler.

Java `this` keyword

In Java, this is a keyword which is **used to refer current object** of a class. we can it to refer any member of the class. It means we can access any instance variable and method by using **this** keyword.

The main purpose of using **this** keyword is to solve the confusion when we have same variable name for instance and local variables.

We can use this keyword for the following purpose.

- **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.

Lets first understand the most general use of this keyword. As we said, it can be used to differentiate local and instance variables in the class.

Example:

In this example, we have three instance variables and a constructor that have three parameters with **same name as instance variables**. Now, we will use this to assign values of parameters to instance variables.

```
class Demo
{
    Double width, height, depth;

    Demo (double w, double h, double d)
    {
        this.width = w;

        this.height = h;

        this.depth = d;
    }

    public static void main(String[] args) {

        Demo d = new Demo(10,20,30);

        System.out.println("width = "+d.width);

        System.out.println("height = "+d.height);

        System.out.println("depth = "+d.depth);

    }
}
```

Copy

```
width = 10.0
height = 20.0
depth = 30.0
```

Here **this** is used to initialize member of current object. Such as, **this.width** refers to the variable of the current object and **width** only refers to the parameter received in the constructor i.e the argument passed while calling the constructor.

Calling Constructor using this keyword

We can call a constructor from inside the another function by using this keyword

Example:

In this example, we are calling a parameterized constructor from the non-parameterized constructor using the this keyword along with argument.

```
class Demo
```

```

{

Demo  ()

{

    // Calling constructor

    this("Studytonight");

}

Demo(String str){

    System.out.println(str);

}

public static void main(String[] args) {

    Demo d = new Demo();

}

}

```

Copy

Studytonight

Accessing Method using this keyword

This is another use of this keyword that allows to access method. We can access method using object reference too but if we want to **use implicit object provided by Java** then use this keyword.

Example:

In this example, we are **accessing getName() method using this** and it works fine as works with object reference. See the below example

```

class Demo

{

    public void getName()

```

```
{

    System.out.println("Studytonight");

}

public void display()

{

    this.getName();

}

public static void main(String[] args) {

    Demo d = new Demo();

    d.display();

}

}
```

Copy

Studytonight

Return Current Object from a Method

In such scenario, where we want to return current object from a method then we can use this to solve this problem.

Example:

In this example, we created a method display that returns the object of Demo class. To return the object, we used this keyword and stored the returned object into Demo type reference variable. We used that returned object to call getName() method and it works fine.

```
class Demo

{

    public void getName()

    {

        System.out.println("Studytonight");

    }

}
```



```
public Demo display()

{

    // return current object

    return this;

}


public static void main(String[] args) {

    Demo d = new Demo();

    Demo d1 = d.display();

    d1.getName();

}

}
```

Copy

Studytonight

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.

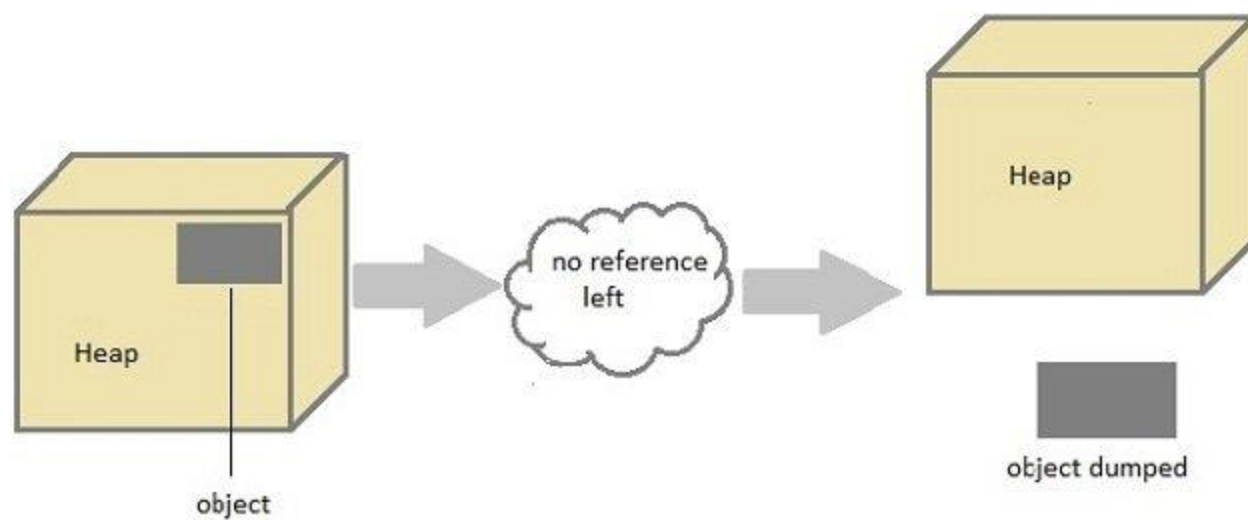
When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called **Garbage Collection**.

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 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.

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
```

Copy

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
```

Copy

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**. Its the last chance for any object to perform cleanup utility.

Signature of **finalize()** method

```
protected void finalize()
{
    //finalize-code
}
```

Copy

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");

    }

}
```

Copy

Garbage Collected

Static Block in Java

Static is a keyword in Java which is used to declare static stuffs. It can be used to create variable, method block or a class.

Static variable or method can be accessed without instance of a class because it belongs to class. Static members are common for all the instances of the class but non-static members are different for each instance of the class. Lets study how it works with variables and methods.

Static Variables

Static variables defined as a class member can be **accessed without object** of that class. Static variable is **initialized once** and **shared among different objects** of the class. All the object of the class having static variable will have the same instance of static variable.

Note: Static variable is used to represent common property of a class. It saves memory.

Example:

Suppose there are 100 employee in a company. All employee have its 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, then declare company_name field as static

Below we have a simple class with one static variable, see the example.

```
class Employee

{

    int eid;

    String name;

    static String company = "Studytonight";

}
```

```
public void show()

{

    System.out.println(eid + "-" + name + "-" + company);

}


public static void main( String[] args )

{

    Employee se1 = new Employee();

    se1.eid = 104;

    se1.name = "Abhijit";

    se1.show();


    Employee se2 = new Employee();

    se2.eid = 108;

    se2.name = "ankit";

    se2.show();

}

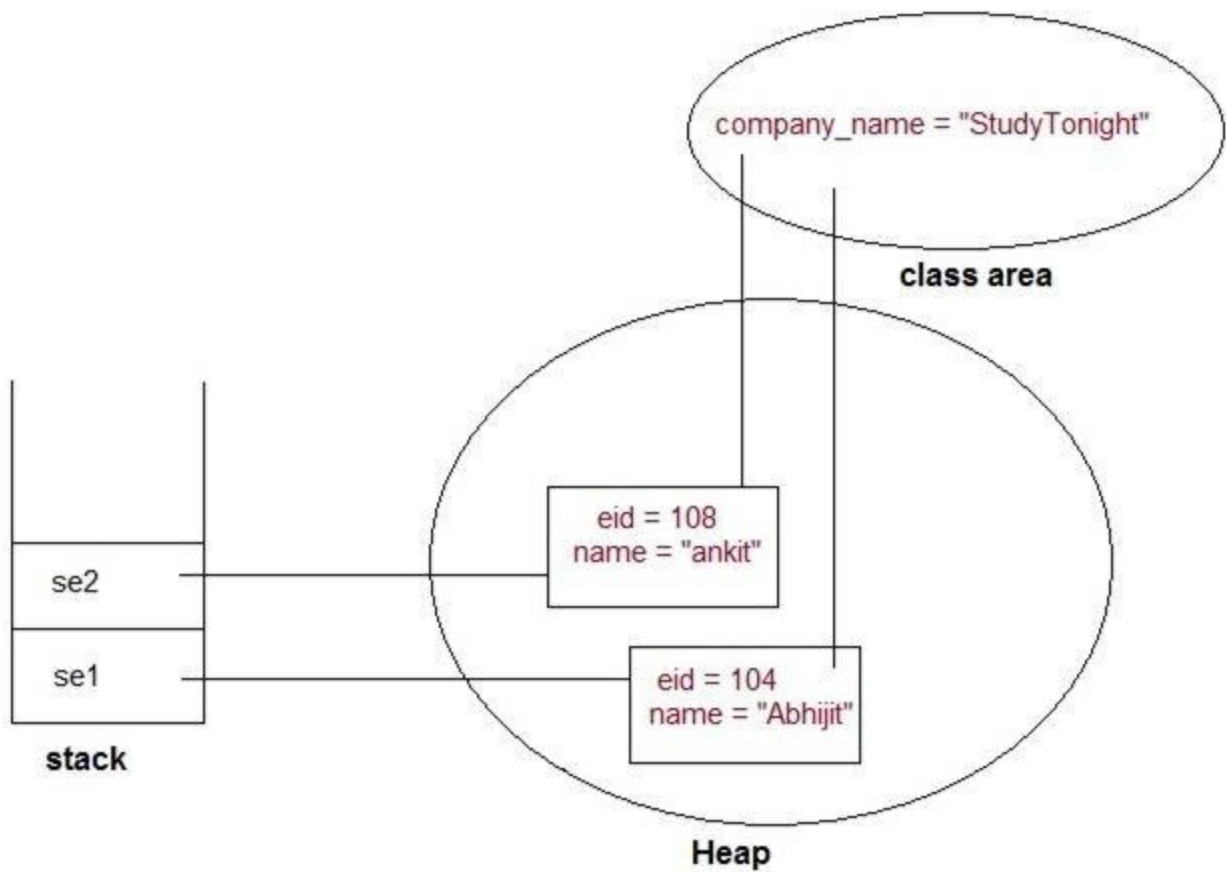
}
```

Copy

104-Abhijit-Studytonight

108-ankit-Studytonight

We can understand it using the below image that shows different memory areas used by the program and how static variable is shared among the objects of the class.



When we have a class variable like this, defined using the **static** keyword, then the variable is defined only once and is used by all the instances of the class. Hence if any of the class instance modifies it then it is changed for all the other instances of the class.

Static variable vs Instance variable

Here are some differences between static variable and instance variables. Instance variables are non-static variables that store into heap and accessed through object only.

Static variable	Instance Variable
Represent common property	Represent unique property
Accessed using class name (can be accessed using object name as well)	Accessed using object
Allocated memory only once	Allocated new memory each time a new object is created

Example: static vs instance variables

Let's take an example and understand the difference.

```
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.

    }

}

```

Copy

101

102

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

Static Method in Java

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.

Let's take an 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.

    }

}

```

Copy

64

Static block in Java

Static block is used to initialize static data members. Static block executes even before main() method.

It executes when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they are programmed.

Example

Lets see an example in which we declared a static block to initialize the static variable.

```

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 sel = new ST_Employee();

        sel.eid = 104;

        sel.name = "Abhijit";

        sel.show();

    }

}

```

Copy

104 Abhijit StudyTonight

Non-static (instance) variable cannot be referenced from a static context.

When we try to access a non-static variable from a static context like main method, java compiler throws an error message 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 we 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

Let's take an example of accessing non-static variable from a static context.

```

class Test

{

    int x;

    public static void main(String[] args)

    {

        x = 10;

    }

}

```

Copy

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

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

Java Final Modifier

Final modifier is used to declare a field as final. It can be used with variable, method or a class.

If we declare a **variable as final** then it **prevents** its **content** from **being modified**. The variable **acts** like a **constant**. Final field **must be initialized** when it is declared.

If we declare a **method as final** then it **prevents** it from being **overridden**.

If we declare a **class as final** the it **prevents** from being **inherited**. We **can not inherit final class** in Java.

Example: Final variable

In this example, we declared a final variable and later on trying to modify its value. But final variable can not be reassigned so we get compile time error.

```
public class Test {

    final int a = 10;

    public static void main(String[] args) {

        Test test = new Test();

        test.a = 15; // compile error

        System.out.println("a = "+test.a);

    }

}
```

Copy

```
error: The final field Test.a cannot be assigned
```

Final Method

A method which is declared using final keyword known as final method. It is useful when we want to prevent a method from overridden.

Example: Final Method

In this example, we are creating a final method learn() and trying to override it but due to final keyword compiler reports an error.

```
class StudyTonight
{
    final void learn()
    {
        System.out.println("learning something new");
    }
}

// concept of Inheritance

class Student extends StudyTonight
{
    void learn()
    {
        System.out.println("learning something interesting");
    }

    public static void main(String args[]) {
        Student object= new Student();

        object.learn();
    }
}
```

Copy

```
Cannot override the final method from StudyTonight
```

This will give a compile time error because the method is declared as final and thus, it cannot be overridden. Don't get confused by the **extends** keyword, we will learn about this in the Inheritance tutorial which is next.

Let's take an another example, where we will have a final variable and method as well.

```
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);

    }
}
```

Copy

In the class above, the MAX_PRICE and MIN_PRICE variables are final hence their values cannot be changed once declared. Similarly the method display() is final which means even if some other class inherits the **Cloth** class, the definition of this method cannot be changed.

Final Class

A class can also be declared as final. A class declared as final cannot be inherited. The String class in **java.lang** package is an example of a final class.

We can create our own final class so that no other class can inherit it.

Example: Final Class

In this example, we created a final class ABC and tried to extend it from Demo class. But due to restrictions the compiler reports an error. See the below example.

```
final class ABC{

    int a = 10;

    void show() {

        System.out.println("a = "+a);

    }

}
```

```
public class Demo extends ABC{

    public static void main(String[] args) {

        Demo demo = new Demo();

    }

}
```

Copy

The type Demo cannot subclass the final class ABC

Java Blank Final Variable

Final variable that is **not initialized** at the time of declaration is called **blank final variable**. Java allows to declare a final variable without initialization but it **should be initialized** by the **constructor** only. It means we can set value for blank final variable in a constructor only.

Example:

In this example, we created a blank final variable and initialized it in a constructor which is acceptable.

```
public class Demo{

    // blank final variable

    final int a;

    Demo(){

        // initialized blank final

        a = 10;

    }

    public static void main(String[] args) {

        Demo demo = new Demo();

        System.out.println("a = "+demo.a);

    }

}
```

```
    }  
}
```

Copy

```
a=10
```

Static Blank Final Variable

A blank final variable declared using static keyword is called static blank final variable. It can be initialized in static block only.

Static blank final variables are used to create static constant for the class.

Example

In this example, we are creating static blank final variable which is initialized within a static block. And see, we used class name to access that variable because for accessing static variable we don't need to create object of that class.

```
public class Demo{  
  
    // static blank final variable  
  
    static final int a;  
  
    static{  
  
        // initialized static blank final  
  
        a = 10;  
  
    }  
  
    public static void main(String[] args) {  
  
        System.out.println("a = "+Demo.a);  
  
    }  
}
```

Copy

a=10

Java instanceof Operator and Downcasting

In Java, **instanceof** is an operator which is used to check object reference. It checks whether the reference of an object belongs to the provided type or not. It returns either **true or false**, if an object reference is of specified type then it return true otherwise false.

We can use instanceof operator to check whether an object reference belongs to parent class, child class, or an interface.

It's also known as type comparison operator because it compares the instance with type.

Application of instanceof

Apart from testing object type, we can use it for object **downcasting**. However, we can perform downcasting using typecasting but it may raise ClassCastException at runtime.

To avoid the exception, we use instanceof operator. Doing this helps to perform casting.

When we do typecast, it is always a good idea to check if the typecasting is valid or not. instanceof helps us here. We can always first check for validity using instanceof, then do typecasting.

Syntax for declaring instanceof operator is given below.

Syntax

```
(object) instanceof (type)
```

Copy

object: It is an object reference variable.

type: It can be a class or an interface.

Time for an Example

Let's take an example to understand the use of **instanceof** operator. Here we are testing the reference type is type of Test class and it returns true.

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

Copy

```
true
```

Example: Interface Reference Type

Lets create an interface and test reference type using instanceof operator. This operator returns if the reference object refers to right interface type. See the below example

```
interface Callable{

    void call();

}

class Demo implements Callable {

    public void call() {

        System.out.println("Calling...");

    }

    public static void main(String[] args) {

        Callable d = new Demo();

        // Refer to a class

        System.out.println((d instanceof Demo));

        // Refer to an interface

        System.out.println((d instanceof Callable));

    }

}
```

Copy

```
true
```

```
true
```

If reference variable holds **null** then instanceof returns **false**. Lets see one more example.


```
class Demo {

    public static void main(String[] args) {

        Demo d = null;

        System.out.println(d instanceof Demo);

    }

}
```

Copy

```
false
```

Example : Inheritance

Lets take another example to understand **instanceof** operator, in which we created 5 classes and some of them extends to another class. Now by creating objects we are testing reference object belongs to which class.

```
class Parent{}

class Child1 extends Parent{}

class Child2 extends Parent{}

class Demo

{

    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

    }

}

```

Copy

```

true
true
false
false
true
false
false
true

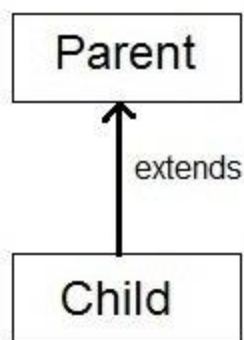
```

Downcasting in Java

Downcasting is one of the advantages of using **instanceof** operator. Downcasting is a process to **hold object** of **parent** class in **child** class **reference** object. It is reverse of **upcasting**, in which **object** of child is assigned to **parent** class reference object.

However, we can perform downcasting using **typecasting** but it throws **ClassCastException** at run-time. So **to avoid** this we use instanceof operator to perform downcasting.

You can understand whole process by the below image.



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);

}

}
```

Copy

Sucessfull Casting

Java Package

Package is a collection of related classes. Java uses package to group related [classes](#), [interfaces](#) and [sub-packages](#) in any Java project.

We can assume package as a folder or a directory that is used to store similar files.

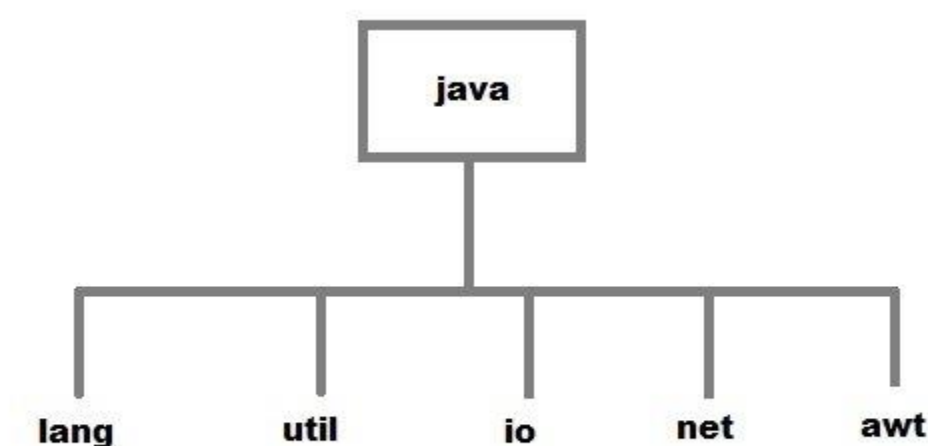
In Java, packages are used to avoid name conflicts and to control access of class, interface and enumeration etc. Using package it becomes easier to locate the related classes and it also provides a good structure for projects with hundreds of classes and other files.

Lets understand it by a simple example, Suppose, we have some math related classes and interfaces then to collect them into a simple place, we have to create a package.

Types Of Java Package

Package can be built-in and user-defined, Java provides rich set of built-in packages in form of API that stores related classes and sub-packages.

- **Built-in Package:** math, util, lang, i/o etc are the example of built-in packages.
- **User-defined-package:** Java package created by user to categorize their project's classes and interface are known as user-defined packages.



How to Create 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
{
    String empId;

    String name;
}
```

Copy

The above statement will create a package with name **mypack** in the project directory.

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

Additional points about package:

- Package statement must be first statement in the program even before the import statement.
- A package is always defined as a separate folder having the same name as the package name.
- Store all the classes in that package folder.
- All classes of the package which we wish to access outside the package must be declared public.
- All classes within the package must have the package statement as its first line.
- All classes of the package must be compiled before use.

Example of Java packages

Now let's understand package creation by an example, here we created a **learnjava** package that stores the FirstProgram class file.

```
//save as FirstProgram.java

package learnjava;

public class FirstProgram{

    public static void main(String args[]) {

        System.out.println("Welcome to package example");

    }

}
```

Copy

How to compile Java programs inside packages?

This is just like compiling a normal java program. If you are not using any IDE, you need to follow the steps given below to successfully compile your packages:

```
javac -d . FirstProgram.java
```

Copy

The **-d** switch specifies the destination where to put the generated class file. You can use any directory name like **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?

To run the compiled class that we compiled using above command, we need to specify package name too. Use the below command to run the class file.

```
java learnjava.FirstProgram
```

Copy

```
Welcome to package example
```

After running the program, we will get “Welcome to package example” message to the console. You can tally that with print statement used in the program.

How to import Java Package

To import java package into a class, we need to use java **import** keyword which is used to access package and its classes into the java program.

Use import to access 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 any class that is present in a different package:

1. without import the package
2. import package with specified class
3. import package with all classes

Lets understand each one with the help of example.

Accessing package without import keyword

If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible. For this approach, there is no need to use the **import** statement. But you will have to use the fully qualified name every time you are accessing the class or the interface. This is generally used when two packages have classes with same names. For example: **java.util** and **java.sql** packages contain **Date** class.

Example

In this example, we are creating a class A in package pack and in another class B, we are accessing it while creating object of class A.

```
//save by A.java
```

```

package pack;

public class A {

    public void msg() {

        System.out.println("Hello");

    }

}

//save by B.java

package mypack;

class B {

    public static void main(String args[]) {

        pack.A obj = new pack.A(); //using fully qualified name

        obj.msg();

    }

}

```

Copy

Hello

Import the Specific Class

Package can have many classes but sometimes we want to access only specific class in our program in that case, Java allows us to specify class name along with package name. If we use import **packagename.classname** statement then only the class with name classname in the package will be available for use.

Example:

In this example, we created a class Demo stored into pack package and in another class Test, we are accessing Demo class by importing package name with class name.

```

//save by Demo.java

package pack;

public class Demo {

    public void msg() {

        System.out.println("Hello");

    }

}

```

```
}

//save by Test.java

package mypack;

import pack.Demo;

class Test {

    public static void main(String args[]) {

        Demo obj = new Demo();

        obj.msg();

    }

}
```

Copy

```
Hello
```

Import all classes of the package

If we use **packagename.* statement**, then all the classes and interfaces of this package will be accessible but the classes and interface inside the [sub-packages](#) will not be available for use.

The **import** keyword is used to make the classes of another package accessible to the current package.

Example :

In this example, we created a class First in **learnjava** package that access it in another class Second by using import keyword.

```
//save by First.java

package learnjava;

public class First{

    public void msg() {

        System.out.println("Hello");

    }

}

//save by Second.java

package Java;
```



```
import learnjava.*;

class Second {

    public static void main(String args[]) {

        First obj = new First();

        obj.msg();

    }

}
```

Copy

Hello

Java Sub Package and Static Import

In this tutorial, we will learn about sub-packages in Java and also about the concept of static import and how it is different from normal **import** keyword.

Subpackage in Java

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

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

Note: The standard of defining package is **domain.company.package** e.g. LearnJava.full.io

Example:

In this example, we created a package **LearnJava** and a sub package **corejava** in the Simple.java file.

```
package LearnJava.corejava;

class Simple{

    public static void main(String args[]){

        System.out.println("Hello from subpackage");

    }

}
```

Copy

To compile the class, we can use the same command that we used for package. Command is given below.

```
javac -d . Simple.java
```

Copy

To run the class stored into the created sub package, we can use below command.

```
java LearnJava.corejava.Simple
```

Copy

After successful compiling and executing, it will print the following output to the console.

```
Hello from subpackage
```

Static import in Java

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.

We can import single or multiple static members of any class. To import single static member we can use statement like the below.

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

Copy

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

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

Copy

Example without using static import

This is alternate way to use static member of the class. In this case, we don't need to use import statement rather we use direct qualified name of the class.

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

Copy

12

Example using static import

In this example, we are using import statement to import static members of the class. Here, we are using * to import all the static members.

```
import static java.lang.Math.*;

public class Test

{

    public static void main(String[] args)

    {

        System.out.println(sqrt(144));

    }

}
```

Copy

12

Java Abstract class and methods

In this tutorial, we will learn about abstract class and methods in Java along with understanding how we can implement abstraction using abstract classes. We will also have some code examples.

Abstract Class

A class which is declared using **abstract keyword** known as **abstract class**. An abstract class may or may not have abstract methods. We cannot create object of abstract class.

It is used to achieve abstraction but it does not provide 100% abstraction because it can have concrete methods.

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

Syntax:

```
abstract class class_name { }
```

Copy

Abstract method

Method that are declared without any body within an abstract class are called **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.

Syntax:

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

Copy

Points to Remember about Abstract Class and Method

Here are some useful 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 may or may not have an abstract method. But if any class has even a single abstract method, then it must be declared abstract.
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.

Example of Abstract class

Let's take an example of the Abstract class and try to understand how they can be used in Java.

In this example, we created an abstract class A that contains a method `callme()` and Using class B, we are extending the abstract class.

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

class B extends A
{
    void callme()
    {
        System.out.println("Calling...");
    }

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

```
        b.callme();

    }

}
```

Copy

```
calling...
```

Abstract class with non-abstract method

Abstract classes can also have non abstract methods along with abstract methods. But remember while extending the class, provide definition for the abstract method.

```
abstract class A

{

    abstract void callme();

    public void show()

    {

        System.out.println("this is non-abstract method");

    }

}

class B extends A

{

    void callme()

    {

        System.out.println("Calling...");

    }

    public static void main(String[] args)

    {

        B b = new B();

        b.callme();

        b.show();

    }

}
```

Copy

calling...

this is non-abstract method

Abstraction using Abstract class

Abstraction is an important feature of OOPS. It means hiding complexity and show functionality only to the user. 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();

    }

}
```

Copy

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.

Java Interfaces

Interface is a concept which is used to achieve abstraction in Java. This is the only way by which we can achieve full abstraction. Interfaces are syntactically similar to classes, but you cannot create instance of an **Interface** and their methods are declared without any body. It can have When you create an interface it defines what a class can do without saying anything about how the class will do it.

It can have only abstract methods and static fields. However, from **Java 8**, interface can have **default** and **static methods** and from **Java 9**, it can have **private methods** as well.

When an interface inherits another interface **extends** keyword is used whereas class use **implements** keyword to inherit an interface.

Advantages of Interface

- It Support multiple inheritance
- It helps to achieve abstraction
- It can be used to achieve loose coupling.

Syntax:

```
interface interface_name {  
  
    // fields  
  
    // abstract/private/default methods  
  
}
```

Copy

Interface Key Points

- Methods inside interface must not be static, final, native or strictfp.
- All variables declared inside interface are implicitly public, static and final.
- All methods declared inside 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.

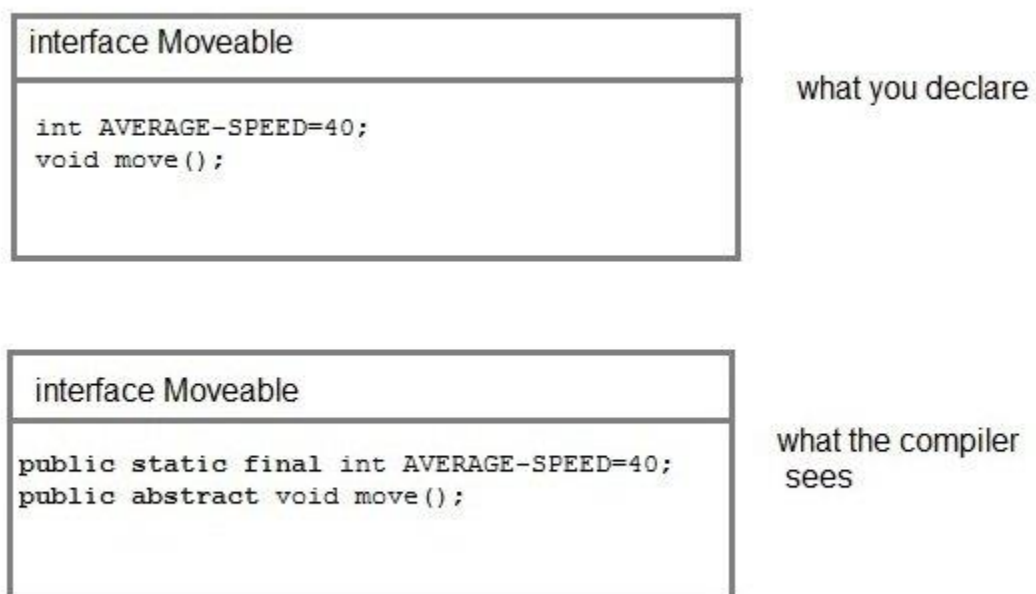
Time for an Example!

Let's take a simple code example and understand what interfaces are:

```
interface Moveable
{
    int AVERAGE-SPEED = 40;

    void move();
}
```

Copy



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

Example of Interface implementation

In this example, we created an interface and implemented using a class. lets see how to implement the interface.

```
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();

    }

}
```

Copy

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 interfaces.

In this example, two interfaces are implemented by a class that show implementation of multiple inheritance.

```
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());

    }

}

```

Copy

true

true

Interface extends other Interface

Interface can inherit to another interface by using extends keyword. But in this case, interface just inherit, does not provide implementation. Implementation can be provided by a class only.

```

interface NewsPaper

{

    news();

}

interface Magazine extends NewsPaper

{

    colorful();

}

```

Copy

Difference between an interface and an abstract class?

Interface and abstract class both are used to implement abstraction but have some differences as well. Some of differences are listed below.

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.

Default Methods in Interface – Java 8

In Java 8 version a new feature is added to the interface, which was default method. Default method is a method that can have its body. It means default method is not abstract method, it is used to set some default functionality to the interface.

Java provides default keyword to create default method. Let's see an example:

```
interface Abc{

    // Default method

    default void msg(){

        System.out.println("This is default method");

    }
```

```

        // Abstract method

        void greet(String msg);
    }

    public class Demo implements Abc{

        public void greet(String msg){           // implementing abstract
method

            System.out.println(msg);

        }

        public static void main(String[] args) {

            Demo d = new Demo();

            d.msg();    // calling default method

            d.greet("Say Hi");    // calling abstract method

        }

    }
}

```

Copy

This is default method

Say Hi

Static methods in Interface – Java 8

From Java 8, Java allows to declare static methods into interface. The purpose of static method is to add utility methods into the interface. In the below example, we have created an interface Abc that contains a static method and an abstract method as well. See the below example.

```

interface Abc{

    // static method

    static void msg(){

        System.out.println("This is static method");

    }

    // Abstract method

    void greet(String msg);

}

```

```

public class Demo implements Abc{

    public void greet(String msg){           // implementing abstract
method

        System.out.println(msg);

    }

    public static void main(String[] args) {

        Demo d = new Demo();

        Abc.msg();    // calling static method

        d.greet("Say Hi"); // calling abstract method

    }

}

```

Copy

This is static method

Say Hi

Private methods – Java 9

In Java 9 version, a new feature is added that allows us to declare private methods inside the interface. The purpose of private method is just to share some task between the non-abstract methods of the interface.

In this example, we created an interface Abc that has a default method and a private method as well. Since private methods are not accessible outside to interface. So, we called it from the default method. See the below example.

```

interface Abc{

    // Default method

    default void msg(){

        greet();

    }

    // Private method

    private void greet() {

        System.out.println("This is private method");

    }

}

```

```
}

public class Demo implements Abc{

    public static void main(String[] args) {

        Demo d = new Demo();

        d.msg();    // calling default method

    }

}
```

Copy

This is private method

Java Nested Class

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

Syntax:

```
class Outer{

//class Outer members

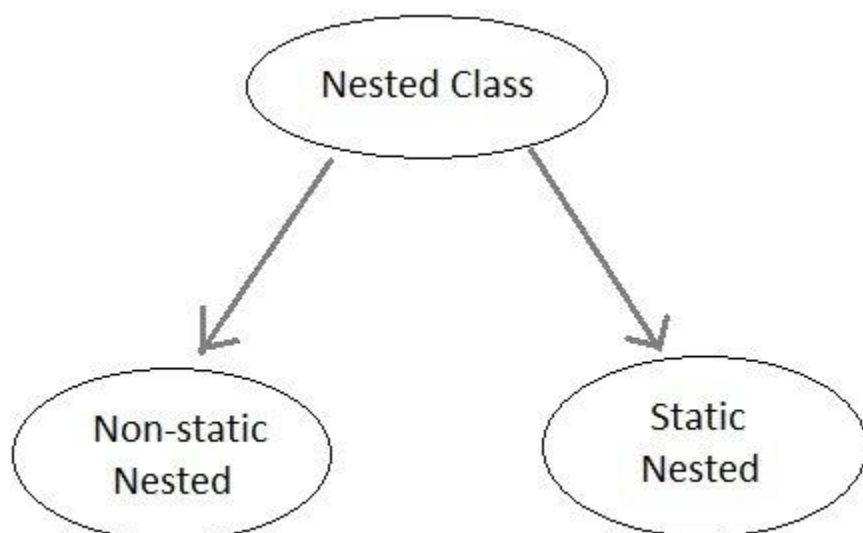

class Inner{

//class Inner members

}

} //closing of class Outer
```

Copy



Advantages of Nested Class

1. It is a way of logically grouping classes that are only used in one place.
2. It increases encapsulation.
3. It can lead to more readable and maintainable code.

If you want to create a class which is to be used only by the enclosing class, then it is not necessary to create a separate file for that. Instead, you can add it as "**Inner Class**"

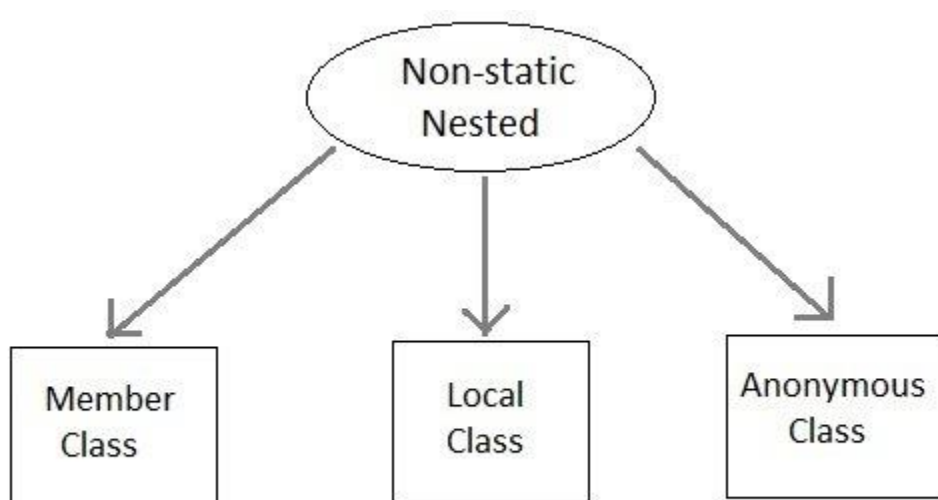
Static Nested Class

If the nested class i.e the class defined within another class, has static modifier applied in it, then it is called as **static nested class**. Since it is, static nested classes can access only static members of its outer class i.e it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested class is rarely used.

Non-static Nested class

Non-static Nested class is the most important type of nested class. It is also known as **Inner class**. It has access to all variables and methods of Outer class including its private data members and methods and may refer to them directly. But the reverse is not true, that is, Outer class cannot directly access members of Inner class. Inner class can be declared private, public, protected, or with default access whereas an Outer class can have only public or default access.

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 directly.



A non-static Nested class that is created outside a method is called **Member inner class**.

A non-static Nested class that is created inside a method is called **local inner class**. If you want to invoke the methods of local inner class, you must instantiate this class inside the method. We cannot use private, public or protected access modifiers with local inner class. Only abstract and final modifiers are allowed.

Example of Inner class(Member 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();

    }

}
```

Copy

Inside inner

Example of Inner class inside a method(local inner class)

```
class Outer

{
```



```
int count;

public void display()

{

    for(int i=0;i<5;i++)

    {

        //Inner class defined inside for loop

        class inner

        {

            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();

    }

}
```

Copy

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();
    }
}
```

Copy

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[])

    {

        //Anonymous class created

        Animal an = new Animal() {

            public void type()

            {

                System.out.println("Anonymous animal");

            }

        };

        an.type();

    }

}
```

Copy

Anonymous animal

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

\Difference Between Concrete Class, Abstract Class, Final Class, Interface.

Class, interface, abstract class, final class are the important component of the Java language. Before discussing about the differences among them first lets get little intro about all these. So that we can get some idea what these terms refer to.

Concrete Class

A class that has all its methods implemented, no method is present without body is known as concrete class.

In other words, a class that contains only non-abstract method will be called concrete class.

Abstract Class

A class which is declared as abstract using abstract keyword is known as abstract class. abstract contains abstract methods and used to achieve abstraction, an important feature of OOP programming. Abstract class can not be instantiated.

For more details, you can refer our detailed tutorial. [Click here](#)

Interface

Interface is a blueprint of an class and used to achieve abstraction in Java. Interface contains abstract methods and default, private methods. We can not create object of the interface. Interface can be used to implement multiple inheritance in Java.

For more details, you can refer our detailed tutorial. [Click here](#)

Final class

Final class is a class, which is declared using final keyword. Final class are used to prevent inheritance, since we cannot inherit final class. We can create its object and can create static and non-static methods as well.

For more details, you can refer our detailed tutorial. [Click here](#)

Here in this table we are differentiating class, abstract class, interface etc based on some properties like: access modifier, static, non-static etc.

	Concrete Class	Abstract Class	Final Class	Interface
Constructor	Yes	Yes	Yes	No
Non-static (method)	Yes	Yes	Yes	Yes
Non-static (variable)	Yes	Yes	Yes	No

Access Modifier (by default)	Default	Default	Default	Public
Object Declaration	Yes	Yes	Yes	Yes
Instantiation	Yes	No	Yes	No
Relation	Both (IS-A & HAS-A)	IS-A	HAS-A	IS-A
Final Declarations	May or May not be	May or May not be	May or May not be	Only Final
Abstract Declarations	No	May or May not be	No	Fully Abstract
Inheritance Keyword	Extends	Extends	No Inheritance	Implements
Overloading	Yes	Yes	Yes	Yes
Overriding	No	No	No	No
Super keyword	Yes	Yes	Yes	No
This keyword	Yes	Yes	Yes	Yes
Byte Code	.class	.class	.class	.class
Anonymous Class	No	Yes	No	Yes
Keywords used for declaration	No keyword	Abstract keyword	Final keyword	Interface keyword

Inheritance	Single	Single	No Inheritance	Multiple
Static Variables	Yes	Yes	Yes	Yes

