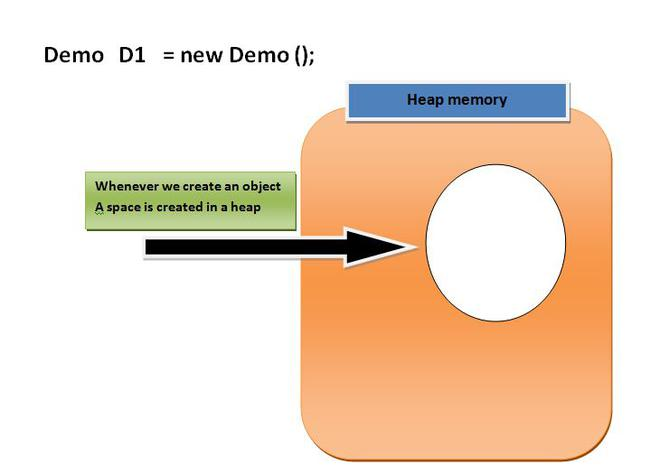
**JAVA PHASE 2**

**REFERENCE VARIABLE**

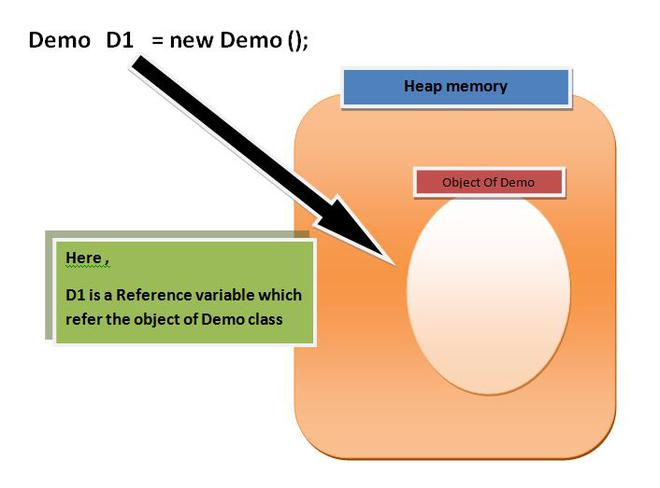
When we create an object (instance) of class then space is reserved in heap memory.

**Demo D1 = new Demo();**

****

Now, The space in the heap Memory is created but the question is how to access that space?.

Then, We create a Pointing element or simply called Reference variable which simply points out the Object(the created space in a Heap Memory).



**Imp points of Reference variable**

1. Reference variable is used to point object/values.

2. Classes, interfaces, arrays etc are reference types in Java. Reference variables hold the objects/values of reference types in Java.

3. Reference variable can also store **null** value. By default, **if no object is passed to a reference variable then it will store a null value.**

4. You can access object members using a reference variable using dot syntax.

**reference variable name .instance variable\_name / method\_name**

**Example**

**D1.display();**

**Or**

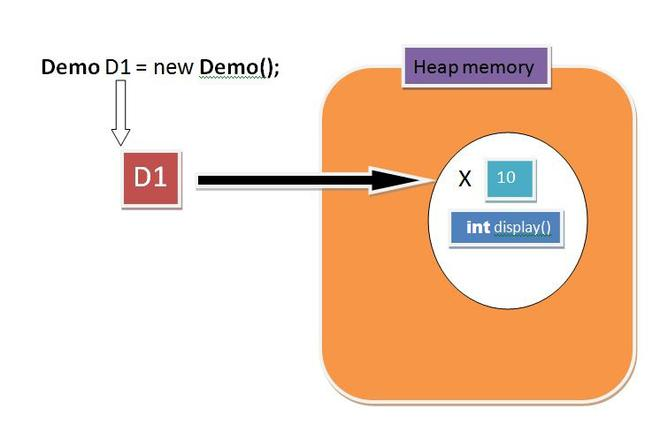
**D1.X;**

Here D1 is reference variable(or object in other term).

display is any function or method

And

X is a variable.



**Oops concepts**

OOPS (object oriented programming System) concepts are as follows:

Class

Objects

Pillars of OOPs:

Abstraction

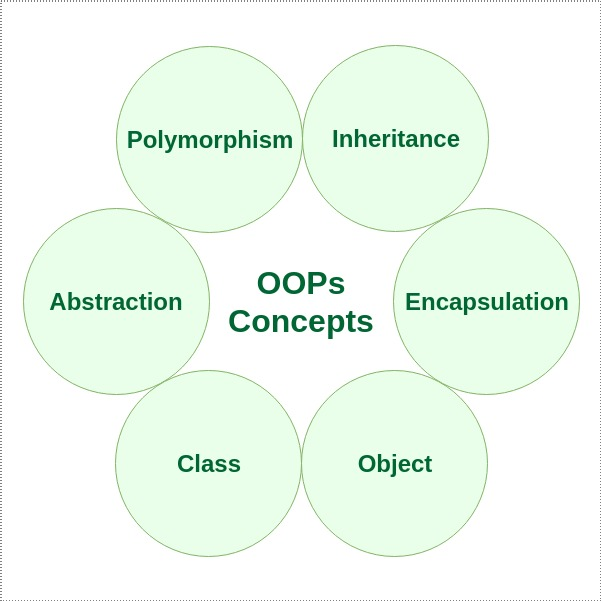
Encapsulation

Inheritance

Polymorphism

Compile-time polymorphism

Runtime polymorphism



**CLASS**

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times.

This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

**Modifiers**: A class can be public or have default access(will discuss later)

**Class name**: The class name should begin with the initial letter capitalized by convention.

**Superclass (if any):** The name of the class’s parent (superclass), if any, preceded by the keyword extends (inheritance).

A class can only extend (subclass) one parent.

**Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

**Body**: The class body is surrounded by braces, { }.

**An object** is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

**State**: It is represented by the attributes of an object. It also reflects the properties of an object.

**Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.

**Identity**: It is a unique name given to an object that enables it to interact with other objects.

**Method**: A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping it, which is why they are considered time savers. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

**public class Itsme**

{

String Employee\_name;

float Employee\_salary;

void set(String n, float p)

{

Employee\_name = n;

Employee\_salary = p;

}

**void get()**

{

System.out.println("Employee name is: " +Employee\_name );

System.out.println("Employee CTC is: " + Employee\_salary);

}

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

{

Itsme G=new Itsme();

G.set("Rathod Avinash", 10000.0f);

G.get();

}

}

**Pillar 1: Encapsulation**

It is defined as the wrapping up of data and functions under a single unit. It is the mechanism that binds together the code and the data it manipulates.

Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.

In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms “encapsulation” and “data-hiding” are used interchangeably.

*Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.*

**Pillar 2: Inheritance**

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields(variables) and methods) of another class.

Let us discuss some frequently used important terminologies:

**Superclass**: The class whose features are inherited is known as superclass (also known as base or parent class).

**Subclass**: The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

**Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

**Why use inheritance in java**

1)For Method Overriding (so runtime polymorphism can be achieved).

2)For Code Reusability.

**SYNTAX**

class Subclass-name extends Superclass-name

{

//methods and fields

}

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

Example of two class with only fields(variables) inherited

**class Employee**

{

float salary=40000;

}

class Programmer extends Employee

{

int bonus=10000;

public static void main(String args[])

{

Programmer p=new Programmer();

System.out.println("Programmer salary is:"+p.salary);

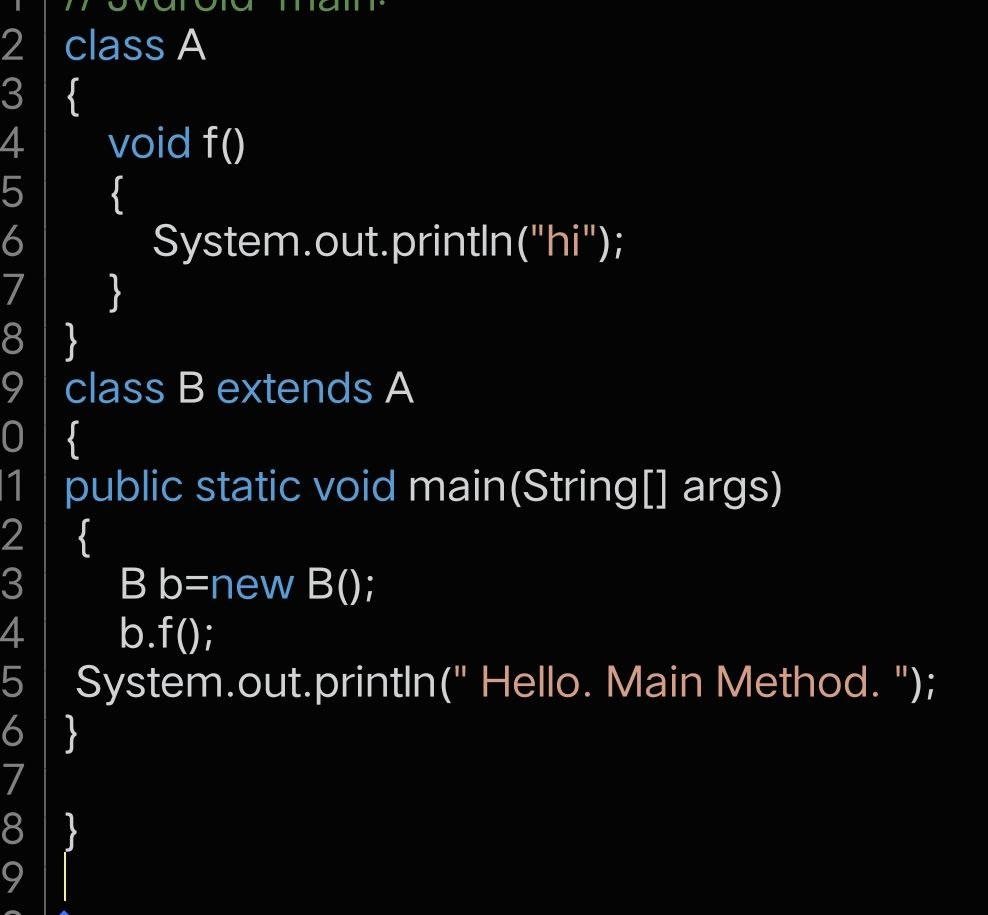
System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

Remember

we only used variables in parent class and child class to explain inheritance but this is not a good way to approach it because data hiding concepts stated that use of variables only done by using the functions.



**Types of inheritance in java**

On the basis of class, there can be three types of inheritance in java:

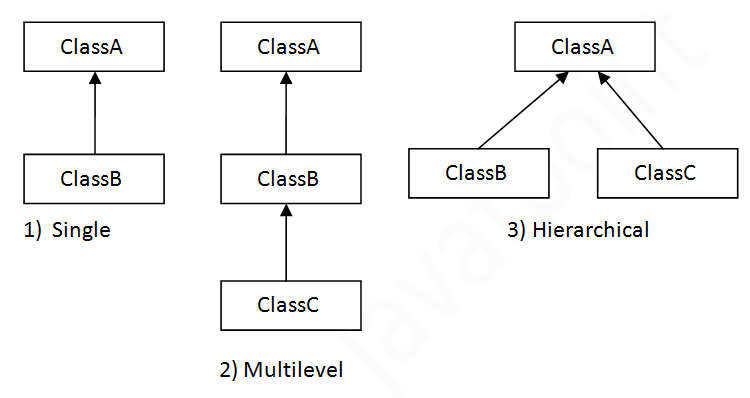
single

multilevel

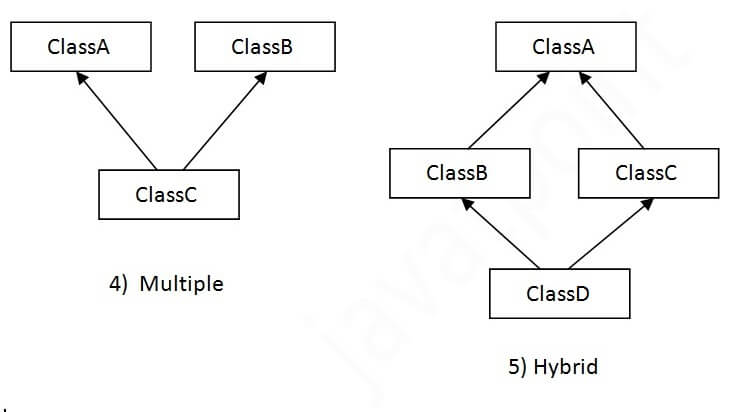
and

hierarchical.

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



Multiple inheritance is not supported in Java through class.it only supported through Interface only.



**Single Inheritance Example**

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

**class Animal**

{

void eat()

{

System.out.println("eating...");}

}

**class Dog extends Animal**

{

void bark()

{

System.out.println("barking...");}

}

class TestInheritance

{

public static void main(String args[])

{

Dog d=new Dog();

d.bark();

d.eat();

}

}

**Multilevel Inheritance**

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

**class Animal**

{

void eat()

{

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

}

}

**class Dog extends Animal**

{

void bark()

{

System.out.println("barking...");}

}

**class BabyDog extends Dog**

{

void weep()

{

System.out.println("weeping...");}

}

**class TestInheritance2**

{

public static void main(String args[])

{

BabyDog d=new BabyDog();

d.weep();

d.bark();

d.eat();

}

}

**Hierarchical Inheritance**

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

**class Animal**

{

void eat()

{

System.out.println("eating...");}

}

**class Dog extends Animal**

{

void bark()

{

System.out.println("barking...");}

}

**class Cat extends Animal**

{

void meow()

{

System.out.println("meowing...");}

}

class TestInheritance3

{

public static void main(String args[])

{

Cat c=new Cat();

c.meow();

c.eat();

//c.bark();//C.T.Error

}

}

**Why multiple inheritance is not supported in java?**

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

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

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

**Example**

**class A**

{

void msg()

{

System.out.println("Hello");

}

}

**class B**

{

void msg()

{

System.out.println("Welcome");}

}

**class C extends A,B**

{

//suppose if it were

public static void main(String args[])

{

C obj=new C();

obj.msg(); //Now which msg() method would be invoked?

}

}

------------------------------

**Polymorphism in Java**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

**Real-life Illustration**:

A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms

**Types of polymorphism**

In Java polymorphism is mainly divided into two types:

**Compile-time Polymorphism**

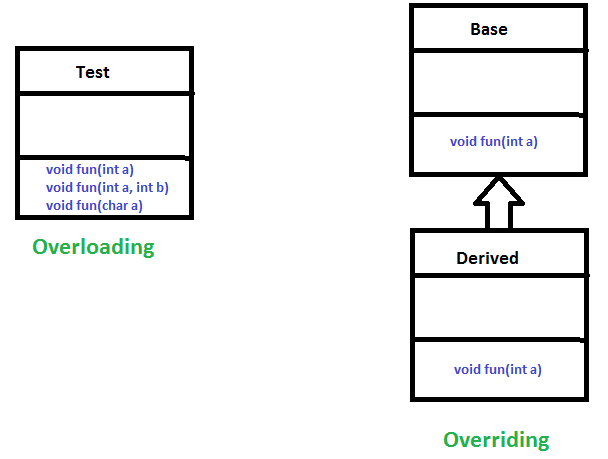
**Runtime Polymorphism**

**Type 1: Compile-time polymorphism**

It is also known as static polymorphism.

This type of polymorphism is achieved by function overloading or operator overloading.

**Note**: But Java doesn’t support the Operator Overloading.



**Method Overloading:** When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

**Example 1**

// Java Program for Method overloading

// By using Different Types of Arguments

**class Helper**

**{**

// Method with 2 integer parameters

static int Multiply(int a, int b)

{

// Returns product of integer numbers

return a \* b;

}

// Method 2

// With same name but with 2 double parameters

static double Multiply(double a, double b)

{

// Returns product of double numbers

return a \* b;

}

}

// Class 2

// Main class

**class GFG**

{

// Main driver method

public static void main(String[] args)

{

/\* Calling method by passing

input as in arguments "/

System.out.println(Helper.Multiply(2, 4));

System.out.println(Helper.Multiply(5.5, 6.3));

}

}

**Output**:

8

34.65

**Example 2**

// Java program for Method Overloading

// by Using Different Numbers of Arguments

**class Helper**

**{**

// Method 1

// Multiplication of 2 numbers

int Multiply(int a, int b)

{

// Return product

return a \* b;

}

// Method 2

// // Multiplication of 3 numbers

int Multiply(int a, int b, int c)

{

// Return product

return a \* b \* c;

}

}

**class GFG**

{

// Main driver method

public static void main(String[] args)

{

// Calling method by passing

// input as in arguments

Helper h=new Helper();

int a=h.multiply(10,20);

Int b=h.multiply (10,20,30);

System.out.println(a);

System.out.println(b);

}

}

**Type 2: Runtime polymorphism**

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is **resolved at Runtime**.

This type of polymorphism is achieved by Method Overriding.

Method overriding, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Example

// Java Program for Method Overriding

**class Parent**

{

// Method of parent class

void Print()

{

// Print statement

System.out.println("parent class");

}

}

**class subclass extends Parent**

{

// Method

void Print()

{

System.out.println("subclass");

}

}

// Main class

**class GFG**

{

// Main driver method

public static void main(String[] args)

{

// Creating object of class 1

subclass a=new subclass();

// Now we will be calling print methods

// inside main() method

a.Print();

}

}

**Output**

Subclass

**Output explanation:**

Here in this program, When an object of child class is created, then the method inside the child class is called. This is because The method in the parent class is overridden by the child class. Since The method is overridden, This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

----------------------

**SUPER KEYWORD**

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

**Usage of Java super Keyword**

super can be used to refer immediate parent class instance variable.

super can be used to invoke immediate parent class method.

super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

**class Animal**

{

String color="white";

}

**class Dog extends Animal**

{

String color="black";

void printColor()

{

System.out.println(color);

//prints color of Dog class

System.out.println(super.color);

//prints color of Animal class

}

}

**class TestSuper1**

**{**

public static void main(String args[])

{

Dog d=new Dog();

d.printColor();

}

}

**Output**

black

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

**class Animal**

{

void eat()

{

System.out.println("eating...");}

}

**class Dog extends Animal**

{

void eat()

{

System.out.println("eating bread...");}

void bark()

{

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

}

void work()

{

super.eat();

bark();

eat();

}

}

**class TestSuper2**

{

public static void main(String args[]){

Dog d=new Dog();

d.work();

}

}

**Output**:

eating...

barking…

eating bread...

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

**class Animal**

{

Animal()

{

System.out.println("animal is created");

}

}

**class Dog extends Animal**

{

Dog()

{

super();

//must be very first statement in case of constructor

System.out.println("dog is created");

}

}

**class TestSuper3**

{

public static void main(String args[])

{

Dog d=new Dog();

}

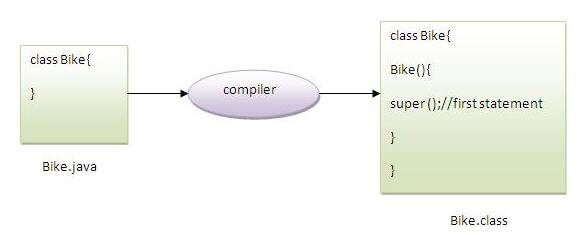
}

**Output**:

animal is created

dog is created

**Note: super() is added in each class constructor automatically by compiler if there is no super().**

****

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

**Another example of super keyword where super() is provided by the compiler implicitly.**

**class Animal**

{

Animal()

{

System.out.println("animal is created");}

}

**class Dog extends Animal**

{

Dog()

{

System.out.println("dog is created");

}

}

**class TestSuper4**

{

public static void main(String args[])

{

Dog d=new Dog();

}

}

**Output**

animal is created

dog is created

-------------+++---------

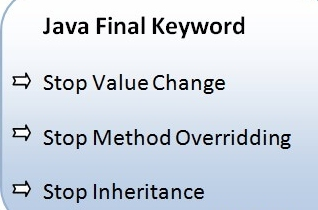
**Final Keyword**

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

class

method

variable



**1) Java final class**

If you make any class as final, you cannot extend it.

**Example**

**final class Bike**

{

void f()

{

Code;

}

}

**class Honda1 extends Bike**

{

void run()

{

System.out.println("running safely with 100kmph");

}

public static void main(String args[])

{

Honda1 honda= new Honda1();

honda.run();

}

}

COMPILE TIME ERROR

**2) Java final method**

If you make any method as final, you cannot override it.

Example of final method

**class Bike**

{

final void run()

{

System.out.println("running");}

}

**class Honda extends Bike**

{

void run()

{

System.out.println("running safely with 100kmph");}

public static void main(String args[])

{

Honda honda= new Honda();

honda.run();

}

}

Compile time error will generate

**Important point**

**Is final method inherited?**

Ans) Yes, final method is inherited but you cannot override it.

**3) Java final variable**

If you make any variable as final, you cannot change the value of final variable(It will be constant).

**Example of final variable**

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

**class Bike9**

**{**

final int speedlimit=90;

//final variable

void run()

{

speedlimit=400;

}

public static void main(String args[])

{

Bike9 obj=new Bike9();

obj.run();

}

}

**Output:Compile Time Error**

**Important point**

What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

**Than question is Can we initialize blank final variable later ?**

Yes, but only in constructor. For example:

**class Bike10**

{

final int speedlimit;

//blank final variable

Bike10()

{

speedlimit=70;

System.out.println(speedlimit);

}

public static void main(String args[])

{

new Bike10();

}

}

**Output: 70**

**Remember**

We can't initialize blank final variable (instance variable) through normal functions.it can only be initialized using user defined constructor.

Means default constructor running by system can't help either.

**What is final parameter?**

If you declare any parameter as final, you cannot change the value of it.

**class Bike11**

{

int cube(final int n)

{

n=n+2;

//can't be changed as n is final

n\*n\*n;

}

public static void main(String args[])

{

Bike11 b=new Bike11();

b.cube(5);

}

}

**Output: Compile Time Error**

**Can we declare a constructor final?**

No, because constructor is never inherited.

--------------++++++--------------

**Abstraction**

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

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

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

**Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

Abstract class (0 to 100%)

Interface (100%)

**Abstract class in Java**

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

**Points to Remember**

An abstract class must be declared with an abstract keyword.

It can have abstract and non-abstract methods.

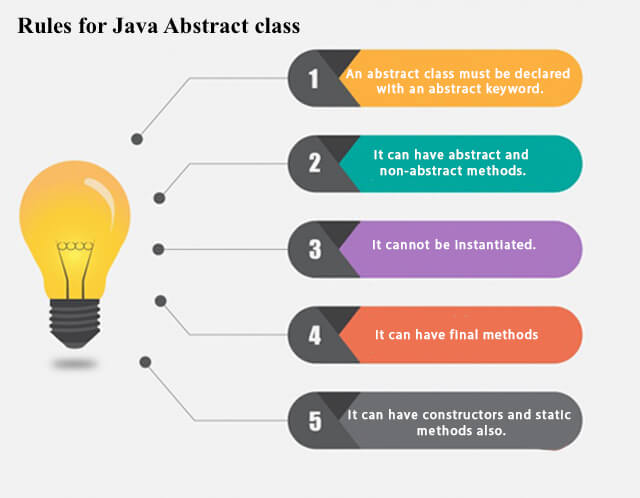
It cannot be instantiated.

Means An instance of an abstract class can not be created.

***Though reference variable can be created (think about it)***

It can have constructors and static methods also.

It can have final methods which will force the subclass not to change the body of the method.



**Abstract Method in Java**

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

abstract void printStatus();

//no method body

**Example of Abstract class that has an abstract method**

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

**abstract class Bike**

{

abstract void run();

}

**class Honda4 extends Bike**

{

void run()

{

System.out.println("running safely");

}

public static void main(String args[])

{

Bike obj = new Honda4();

obj.run();

}

}

**Output : running safely**

**Another real life example of Abstract class in java**

**abstract class Bank**

{

abstract int getRateOfInterest();

}

**class SBI extends Bank**

{

int getRateOfInterest()

{

return 7;

}

}

**class PNB extends Bank**

{

int getRateOfInterest()

{

return 8;

}

}

**class TestBank**

{

public static void main(String args[])

{

Bank b;

b=new SBI();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

b=new PNB();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

}

}

**Output:**

**Rate of Interest is: 7 %**

**Rate of Interest is: 8 %**

**Remember:**

**An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method**

**Rule 1: If there is an abstract method in a class, that class must be abstract.**

But remember

We can have an abstract class without any abstract method.

**Rule 2: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this (child) class abstract.**

**(Think why !!!)**

Means If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method.

**Rule 3:There can be a final method in abstract class.**

**but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error:**

**“Illegal combination of modifiers: abstract and final”.**

**(Think why !!!)**

//another real life example

// Java Program to Illustrate Abstract Class

// Can contain Constructors

// Class 1

// Abstract class

**abstract class Base**

{

// Constructor of class 1

Base()

{

// Print statement

System.out.println("Base Constructor Called");

}

// Abstract method inside class1

abstract void fun();

}

// Class 2

**class Derived extends Base**

{

// Constructor of class2

Derived()

{

System.out.println("Derived Constructor Called");

}

// Method of class2

void fun()

{

System.out.println("Derived fun() called");

}

}

// Class 3

// Main class

**class GFG**

{

// Main driver method

public static void main(String args[])

{

// Creating object of class 2

// inside main() method

Derived d = new Derived();

d.fun();

}

}

**Output**

**Base Constructor Called**

**Derived Constructor Called**

**Derived fun() called**

-------------++++++------------

**INTERFACE**

An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction.There can be only abstract methods in the Java interface, not the method body.

It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also represents the IS-A relationship.

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

1) Interfaces specify what a class must do and not how. It is the blueprint of the behaviour.

2)Interface do not have constructor.

3)If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

**Syntax**:

**interface** interface\_name

{

// declare constant fields

// declare methods that abstract

// by default.

}

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default.

A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

**Why do we use an Interface?**

It is used to achieve total abstraction.

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

It is also used to achieve loose coupling.(search it)

Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

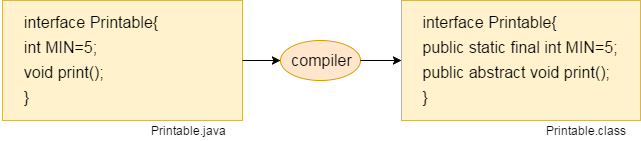
**Remember**

**Internal addition by the compiler**

The Java compiler adds public and abstract keywords before the interface method.

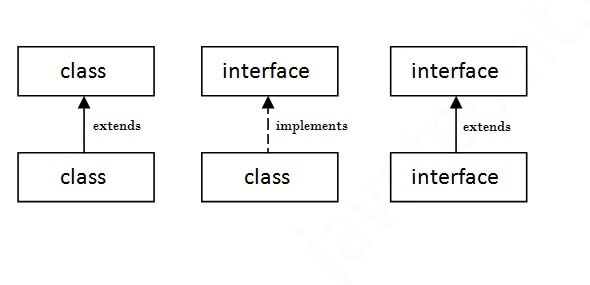
Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



**The relationship between classes and interfaces**

a class extends another class, an interface extends another interface, but a class implements an interface.



**Interface Example**

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

**interface printable**

{

void print();

}

**class A6 implements printable**

{

public void print()

{

System.out.println("Hello");}

public static void main(String args[])

{

A6 obj = new A6();

obj.print();

}

**Remember**

To implement interface method you need to put public keyword during its definition in implementing class.

As shown in previous program.

**Multiple inheritance is not supported through class in java, but it is possible by an interface, why?**

As we have explained in the inheritance, multiple inheritance is not supported in the case of class because of ambiguity.

However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

**For example:**

**interface Printable**

{

void print();

}

**interface Showable**

{

void print();

}

**class TestInterface3 implements Printable, Showable**

{

public void print()

{

System.out.println("Hello");}

public static void main(String args[])

{

TestInterface3 obj = new TestInterface3();

obj.print();

}

}

**Output**:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

**Interface inheritance**

A class implements an interface, but one interface extends another interface.

**interface Printable**

{

void print();

}

**interface Showable extends Printable**

{

void show();

}

**class TestInterface4 implements Showable**

{

public void print()

{

System.out.println("Hello");

}

public void show()

{

System.out.println("Welcome");}

public static void main(String args[])

{

TestInterface4 obj = new TestInterface4();

obj.print();

obj.show();

}

}

**Output:**

**Hello**

**Welcome**

**Java 8 Default Method in Interface**

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

**interface Drawable**

{

void draw();

default void msg()

{

System.out.println("default method");}

}

**class Rectangle implements Drawable**

{

public void draw()

{

System.out.println("drawing rectangle");

}

}

**class TestInterfaceDefault**

{

public static void main(String args[]){

Drawable d=new Rectangle();

d.draw();

d.msg();

}

}

**Output:**

**drawing rectangle**

**default method**

**Java 8 Static Method in Interface**

Since Java 8, we can have static method in interface. Let's see an example:

**interface Drawable**

{

void draw();

static int cube(int x)

{

return x\*x\*x;

}

}

**class Rectangle implements Drawable**

{

public void draw()

{

System.out.println("drawing rectangle");

}

}

**class TestInterfaceStatic**

{

public static void main(String args[])

{

Drawable d=new Rectangle();

d.draw();

System.out.println(Drawable.cube(3));

}

}

**Output:**

**drawing rectangle**

**27**