classmate
Date 11 Aug 2020
Page

Object Oriented Programming Language :- [oop's, pillar of Java, pie mechanism]

- object means real time entity so object-oriented programming is a paradigm to design a program using classes and objects.
- oop's concept have some pillars they are as follows:

1) Inheritence
2) polymorphism
3) Casting / Conversion
4) Abstraction
5) Interface
6) Encapsulation

1) Inheritence :- [for Inheritence keyword = "extends"]

- Inheritence is one of the important pillar in object-oriented programming structure (oops).
- In which one class can aquire the properties of another class with the help of extends keywords that we called it as "Inheritence".
- Also we can say that subclass can aquire the properties of superclass with the extends keyword.
- super class is where the properties are inhering or aquiring so that why we called it as "superclass".
- sub class is where the properties are inherited or delivered that we called as "subclass".
- Inheritence is taken place between two or more classes.
- Inheritence is classified into four types so they are as:

1) Single level Inheritence
2) Multi level Inheritence
3) Multiple Inheritence
4) Hierarchical Inheritence

syntax:- Inheritence

```
class super {
.....
}
class sub extends super{
.....
}
```

1) Single Level Inheritence :-

- For single level Inheritence two classes are mandatory
- It is an operation where inheritence takes place between two classes to perform "single level Inheritence"
- In short we can say that when a class inherits another class, it is known as single Inheritence.
- Diagramatic Representation :-

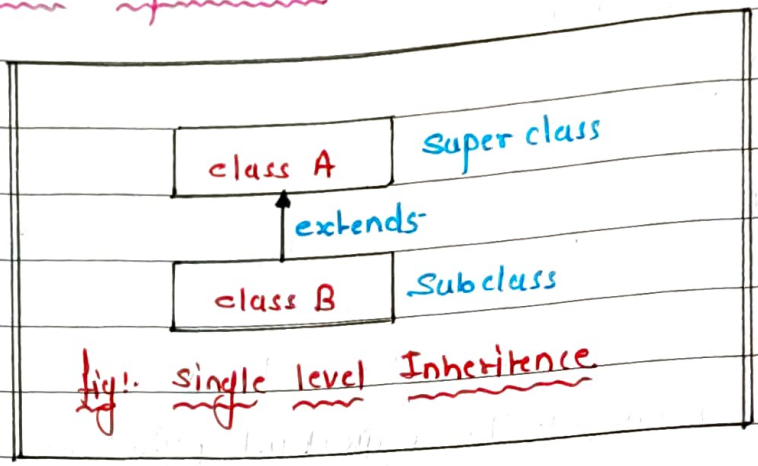


fig:- single level Inheritence

- syntax :-

```
public class A {
    .....
}
public class B extends A {
    .....
}
```

- Example :-

```
class A { public class Animal {
        public void eat() {          // super class
            System.out.println ("Eating");
        }

class B { public class Dog extends Animal {    // sub class with extends
        public void bark() {
            system.out.println ("Barking");
        }
```

```
public class TestInheritence {   // main Test class
    public static void main (String [] args) {
        Dog d = new Dog();
        d.bark();
        d.eat();
    }
}
```

main class

output:- Barking
Eating

Note:- In this we prepare 3 different classes.
class A, B, TestInheritence.

example :- [Single level Inheritence]

1) suppose we have a super class of Name with "BankingFeature" and in that there is a method with method name is "Account".
2) For subclass we have class name with SBI.
3) so in subclass i.e in SBIclass need to aquire the properties of superclass i.e "Account" method in it with the "extends" keyword.
4) so this is a single level Inheritence.
5) In main method we need to create object for subclass and then we are able to call all the methods which is present in superclass.

```
public class BankingFeature {
        public void Account() {
            System.out.println("you have an Account in SBI");
        }
    public class SBI extends BankingFeature {
            public void ATMmachine() {
                System.out.println("your SBI ATM card is active");
            }
    public class Test {
        public static void main (String [] args) {
            SBI s = new SBI();
```

```
        s. AtmMachine ();
        s. Account ();
    }
}
```

output:- your SBI ATM card is Active.
you have an Account in SBI.

2) Multilevel Inheritence :-

- Multilevel Inheritence takes place between three or more classes.
- In multiple Inheritence one subclass can aquire the property of another superclass and this phenomenon continoue so that we known as "Multilevel Inheritence."
- In short when there is a chain of Inheritence is also called as multilevel Inheritence.
- Diagramatic Representation :-

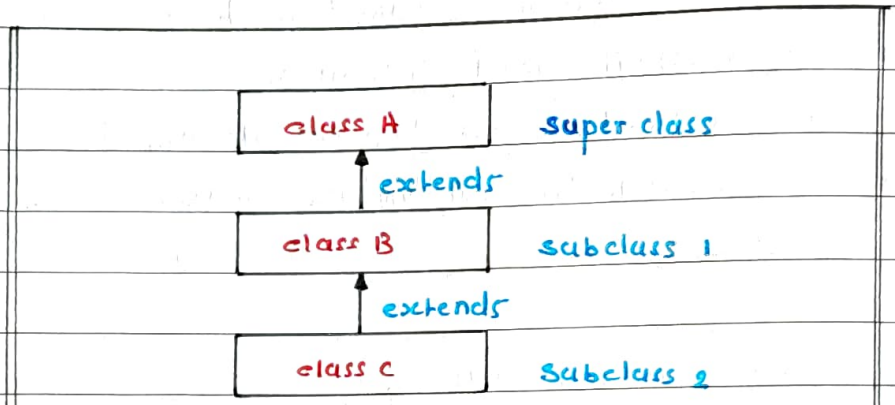


fig:- Multilevel Inheritence

- syntax :-

```
public class A {
    ..... }
public class B extends A {
    ..... }
public class C extends B {
    ..... }
```

- example:-

```
public class Animal {
        public void eat() {
            system.out.println("Eating");
        }
public class Dog extends Animal {
        public void bark() {
            system.out.println("Barking");
        }
public class Babydog extends Dog {
        public void weep() {
            system.out.println("weeping");
        }
public class Test {
    public static void main(string[] args) {
        Babydog d = new Babydog();
        d.weap();
        d.bark();
        d.eat();
    }
}
```

class 1 | class 2 | class 3 | Test class

output:- weeping
Barking
Eating

Note:- In this we have to make three different class and 1 test class for calling methods.

3) Multiple Inheritence :-

- In multiple Inheritence one subclass aquiring property of two superclass at a same time so this is known as "Multiple Inheritence".
- Java doesn't support multiple Inheritence because it results in diamond ambuiguity problem.
- It means the diamond like structure get created in JVM and object Variable get confused for who need to inherited the property of superclass into subclass.

- Because super class of all the classes is object class. So there diamond ambiguity forms.
- How diamond Ambiguity form :-
  In this we have one subclass which aquire the properties of two superclass so it becomes "down" "V" and As we know there is object class in Java for all classes so "upper" V is created so there will be diamond form.
- To reduce the complexity and simplify the language multiple inheritence is not supported in Java.
- (Diamond) Diagramatic Representation :-

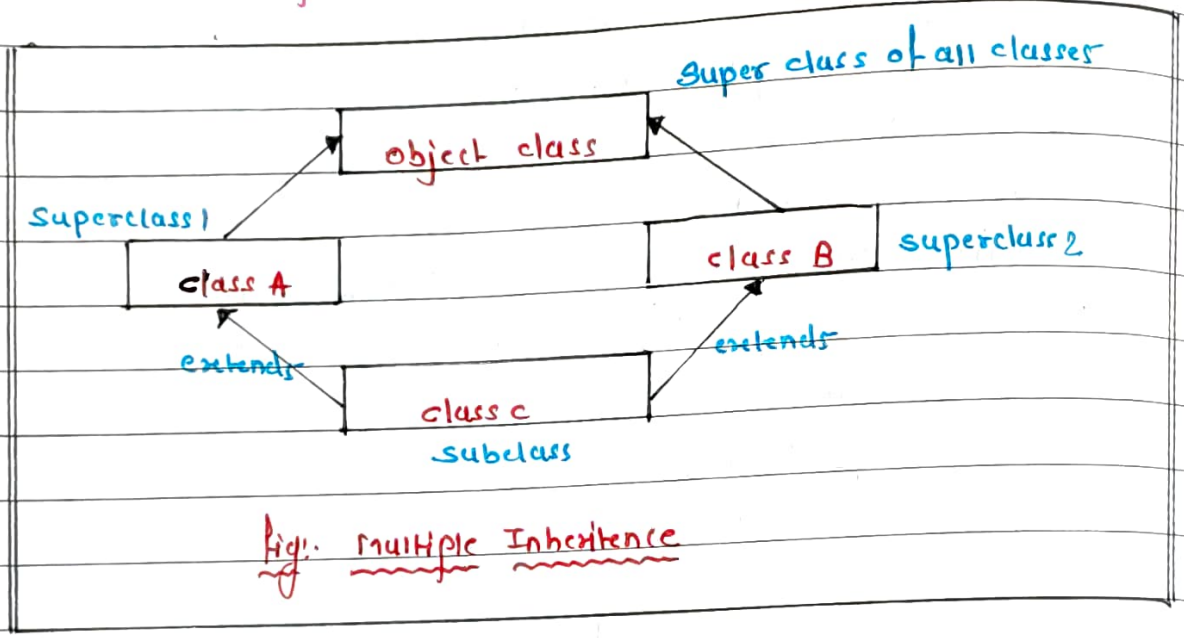


Fig:- Multiple Inheritence

- Syntax :-

```
public class A {
        ..... }

public class B {
        .... }

public class c extends A, B {
        .... }

// Java does not support Multiple Inheritence
```

- example :-

```
class A
public class A {
    public void msg {
        System.out.println ("Hello");
    }

class B
public class B {
    public void msg{
        System.out.println ("welcome");
    }

class C
public class C extends A, B {
    public static void main (String [] args) {
        C obj = new C();
        obj.msg(); // Now which msg() method would
                   be invoice?
    }
}
```

output:- compile time error

4) Hierarchical Inheritence :-

- Hierarchical Inheritence takes place between one super class and multiple sub class.
- so the property of superclass can be aquired by multiple subclass this is known as "Hierarchical Inheritence".
- When two or more classes inherites a single class it is known as Hierarchical Inheritence.
- This Inheritence takes place between one super class and multiple sub class.
- Diagramatic Representation on Next page:.....

Diagramatic Representation :-

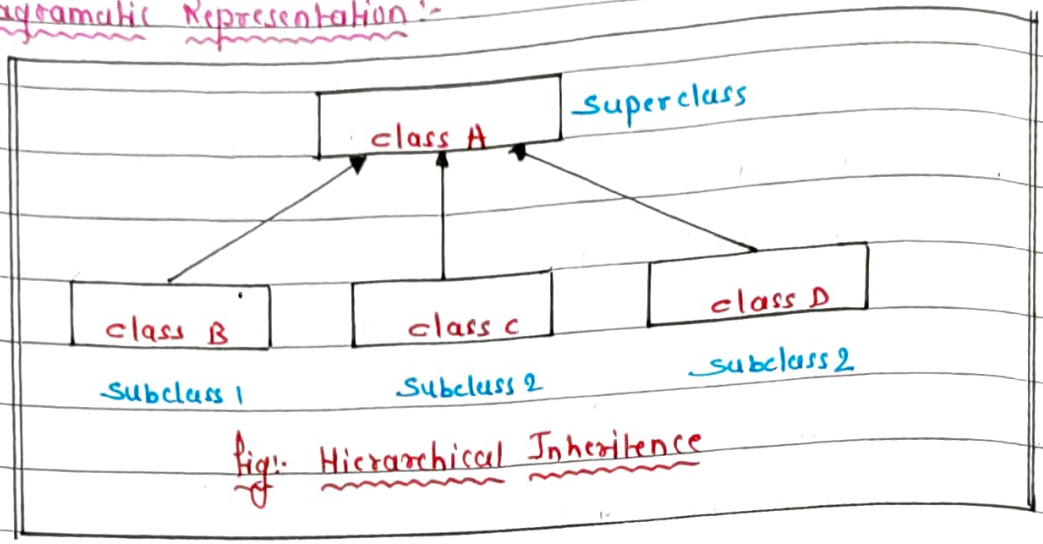


Fig:- Hierarchical Inheritence

- syntax :-

```
public class A {
    ----- }
public class B extends A {
    ----- }
public class C extends A {
    ----- }
```

- example :-

class 1 Superclass

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

class 2 subclass 1

```
public class Dog extends Animal {
    public void bark() {
        system.out.println("barking");
    }
```

class 3 subclass 2

```
public class cat extends Animal {
    public void meow() {
        System.out.println("meowing");
    }

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

```
    Cat c = new Cat();
    c.meow();
    c.eat();
}
```

output:- meowing
eating

2) Polymorphism :-

- polymorphism in Java is a concept by which we can perform a single action in different ways.
- ploymorphism is derived from 2 Greek words: poly and morphs. the word "poly" means many and "morphs" means forms.
- one object shows different behaviour at different stages of life cycle is known as "polymorphism".
- There are two types of polymorphism in Java:
  1) Compile Time polymorphism
  2) Run Time polymorphism

1) Compile Time Polymorphism :-

- In compile time polymorphism method declaration is going to get binded to it's definition at compile time based on Argument. so that why we called it as "Compile time polymorphism".
- At compile-time, java knows which method to call by checking the method signatures, so this is called compile time polymorphism or "static" or "Early binding".
- Compile time polymorphism is achieved through "method overloading".
- As binding takesplace during compilation time so it is known as "Early binding".
- Method overloading is an example of compile time polymorphism.

* Run Time polymorphism :-

- In Runtime polymorphism method declaration is going to get binded to its definition during run time or execution time based on object creation so thats why there is "Runtime polymorphism".
- method overriding is an example of run time polymorphism.
- Runtime polymorphism also called as "dynamic binding" or "Late binding".

* method overloading :-

- Declaring multiple method with same method name but with different arguments in a same class is known as "method overloading".
- example :-

class 1 main

```
public class overloading {
    public void addition (int a, int b) {   // same method name with 2 argument.
        int c = a+b;
        system.out.println (c);
    }
    public void addition (int a, int b, int c) {   // same method name with 3 argument.
        int d = a+b+c;
        system.out.println (d);
    }
}
```

class 2 test

```
public class demo {
    public static void main (string [] args) {

        overloading o = new overloading ();
        o.addition (2,3);
        o.addition (1,3,5);
    }
}
```

output:- 5
9

* Method overriding :-

- Aquiring superclass method into subclass with the help of extends keyword and changing implementation according to subclass specification that is called as "method overriding".

- example :-

class1
```
public class Vehicle {
    public void run () {
        System.out.println (" Vehicle is running");
    }
```

class2
```
public class Bike2 extends Vehicle {
    public void run() {
        System.out.println (" Bike is running safely");
    }
```

Test class
```
public static void main (String [] args) {
    Bike2 obj = new Bike2 ();
    obj.run();
}
}
```

output:- Bike is running safely

Note:- method overloading and method overriding are the best example for compile time and Runtime polymorphism.

8) Casting / conversions :-

- Converting one type of Information into another type of information is known as "casting".
- There are two typer of casting they are as follows:-

1) primitive Casting
2) Non-primitive casting

1) primitive casting :-

- Converting one type (datatype) information into another datatype information is known as "primitive casting".
- primitive casting we also called it as "Widening / casting" conversion
- This casting takes place when two data types are automatically converted. This happen when
  - The two datatypes are compatible.
- There are Three subtypes of primitive casting are as follows:-

Implicite casting point

  i) Implicite casting / Widening conversion
  ii) Explicite casting / Narrowing conversion
  iii) Boolean casting

1) Implicite casting :- [Automatic Conversion]

- Converting (datatype) Lower datatype into Higher datatype.
- That means converting Lower datatype Information into bigger datatype information is called as "Implicite casting".
- Another Name for Implicite casting is "Widening casting" or "Automatically converted" conversion.
- Implicite casting formate:-

Fig:- Implicite / Widening conversion

- That's why we called as widening casting where memory size goes on increasing.
- example :-

```
public class ImplicitCasting {
    public static void main (String [] args) {
        int a = 10;          // 4-byte data
        double b = a;
        System.out.println(b);
    }}
```

output :- 100

Note :- Here byte → short → int → long will follow this pattern

ii) Explicite casting :- [manual conversion]

- In Explicite Casting converting Higher datatype into Lower datatype is known as "Explicite casting".
- In Explicite casting data loss is takes places, so it will also called as "Narrowing casting" or "mancually conversion".
- If we want to assign a value of larger data type to a smaller datatype we perform explicite casting or narrowing:
  - This is useful for incompatible datatypes where automatic conversion cannot be done.
  - Here, target type specifier the desired type to convert the specified value to.
- Explicite Casting formate :-

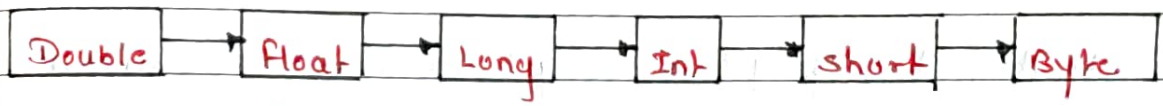


fig :- Explicite / Narrowing Casting

Note :-

- Narrowing Casting must be done manually by placing the type in parantheses () in front of the value.

- example :-

```
public class ExplicitCasting {
    public static void main (String [] args) {
        double a = 10.5 ;   // 8 byte
        int b = (int) a ;
                  ↳ casting
        System.out.println ("b");
    }
}
```

output :- 10

Note :- Here we implement
byte ← short ← int ← long pattern

Q. Write a programe for explicite casting ?

```
public class ExplicitCasting {
    public static void main (String [] args) {
        char ch = 's';
        int num = 99 ;
        ch = num ;
        System.out.println ( ch );
    }
}
```

output :- 7: error incompatible types: possible lossy conversion from int to char
ch = num ;

- In this above programe we get an error because it is incompatible type. so for that we have to write like we have to pass the value in ().

iii) Boolean Casting :-

- It consider to be an incompatible casting type.
- Boolean returns true or false, so for converting true into

false and false into true it is not possible.
- Java doesn't support Boolean casting, so that's why it considers as incompatible type casting.
- It is considered to be as incompatible casting it means we cannot convert true into false it is not possible in java.

2) Non-primitive casting :-
- Non-primitive casting means converting one type of class into another type of class is known as "Non-primitive casting".
- Non-primitive type casting have two subtypes they are as:
  1) Upcasting
  2) Down casting

1) Upcasting :-
- Assigning sub class property into superclass is known as "Upcasting".
- Before performing upcasting operation inheritence operation take place in it.
- After performing inheritence the properties which are present in super class that comes into subclass.
- In subclass we have to declare new properties as well.
- At the time of upcasting the property which are inherited from super class are only eligible for operation.
- The new property which where declared inside subclass are not eligible for upcasting operation.
- example :- (syntax)

```
Parent p = new child(); //syntax
father s = new son();
```

// This is the way we achieve upcasting.

- syntax :- example

script on Next page....

class1 father class

```
public class father {
    public static void main (String [] args) {
        public static void money () {
            System.out.println (" 10000 ");
    }}
```

class2 son class

```
public class Son extends father {
    public static void main (String [] args) {
        public static void money () {
            System.out.println ("5000");
    }}
```

Test class

```
public class Upcasting {
    public static void main (String [] args) {
        father s = new son ();   - upcasting
        s.money ();
    }}
```

output :- 5000

2) Downcasting :-

- Java doesn't support down casting/Sampling before perfor down casting, perform upcasting operation first then perform downcasting.
- Downcasting means the typecasting of a parent object to child object.
- Downcasting cannot be Implicitly.
- In this we can forcefully cast a child to a parent whi is known as "Downcasting".

- syntax :-

```
child c = (child) p;
demo2 d = (demo2) new demo   //example
```

- Downcasting has to be done externally and due to downcasting a child object can acquire the properties of the parent object.

- Diagrame :-

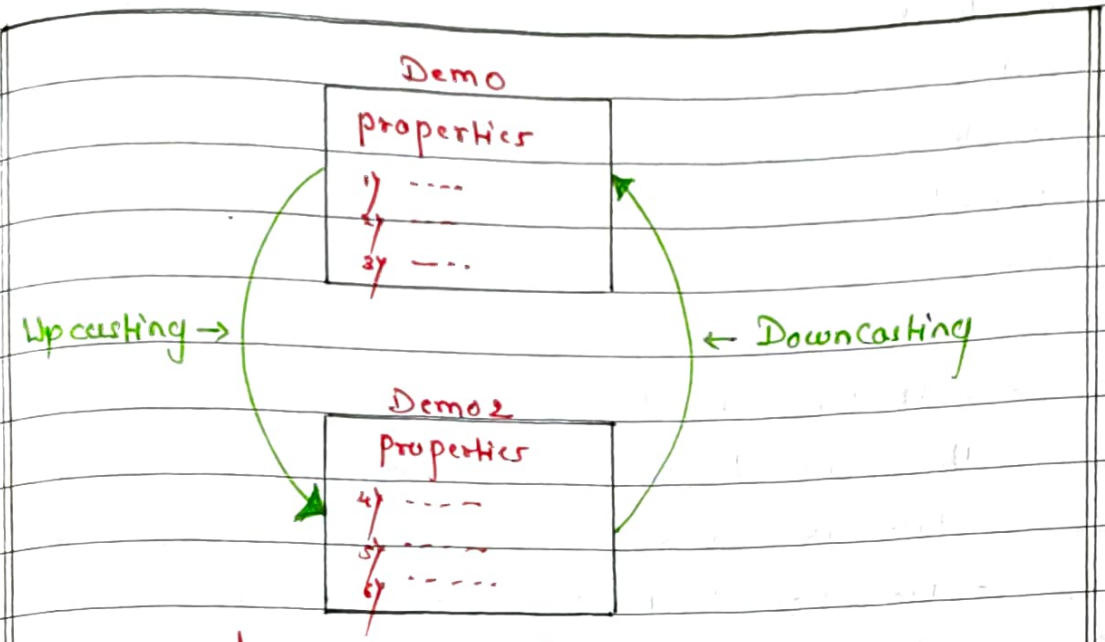
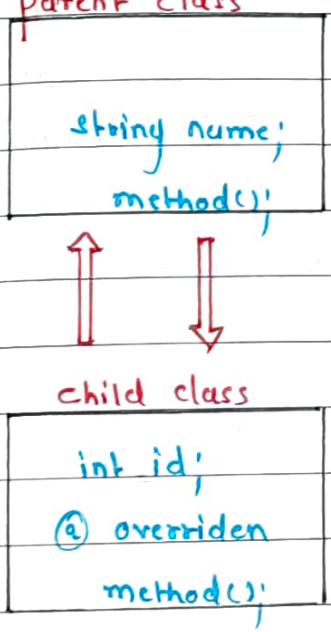


fig:- Upcasting & Downcasting

Note:- Diagrammatic Remember points :-




```
parent p = new child(); //upcasting
p.name = "Velocity";
p.id = 1234; // Not accessible
p.method(); //overriden method
// Trying to Downcasting implicitly
child c = new parent();
// compile time error
child c = (child) p; // Downcasting
                        Explicitly
c.id = 1234;
c.method();
```

fig:- Concept of Upcasting & Downcasting

Class
Date 19 Aug
Page

★ Access specifier :-

- Access specifier are used to represent the scope of mem of class.
- Access specifier are divided into 4 types are as follow
  1) private
  2) Default
  3) protected
  4) public

1) **private** :- If you declared any member of class as private the scope of that member remains only within the class, it can't be access from other class.
   - The access level of private specifier is only within the class.
   - It cannot be accessed from outside of the clas.

2) **Default** :- If you declared any member of a class as defau then scope of that member remains only within the package. it can't be accessed from other packa
   - There is no keyword to represent Default access specifier.
   - The access level of default modifier is only wit the package.
   - If we do not specify any access specifier, it will be the default.

3) **protected** :- If you declared any member of class protected th scope of that member remains within the package.
   - That class which is present outside the package ca also access it by one condition i.e Inheritence operati

- The access level of a protected specifier is within the package and outside the package through child class.
- If we do not make the child class, it cannot be accessed from outside the package.

4) public :- If you declared any member of class as public then scope of that member remains through out the project.
- The access level of public modifier is every where. It can be accessed from within the class, outside the class, within the package and outside the package.

4) Abstraction :-
- Abstraction is one of the pillar of object-oriented programming language (i.e oops) principle.
- Abstraction is a process of Hiding the implementation code and providing only functionality to the end user is known as "Abstraction".
- example:-
We sending sms where we type the text and send the message. but we don't know the internal processing about the message delivary.
- There are two ways to achieve abstraction in java are as
  1) Abstract class (0 to 100%)
  2) Interface (100%)

1) Abstract class in Java:-
- A class which is declared as "abstract" keyword is known as "Abstract class".
- An abstract class means, it having an incomplete class where programmer can declare complete as well as incomplete method.
- In short it can have abstract and Non-abstract methods.

- An incomplete method / ~~Non~~ abstract method means which method declaration is present but method definition be absent.
- we are not able to create an object of abstract class.
- we can declared incomplete method as abstract method declaring keyword abstract infront of method.
- syntax for declaring "abstract class" :-

```
Abstract class Velocity {
    public void test() {
        ---- }          // complete method

abstract class {
    abstract public void test2();   // Incomplete method
    abstract public void test3();
}
```

- point to rember :-
  1) An abstract must be declared with an abstract keyword.
  2) It can have Complete and Incomplete methods.
  3) We are not able to create object of abstract class.
  4) It can have constructor and static method.
  5) It can have final methods which will force the subclass not to change the body of the method.

Concrete class :-

- We can't create object of ~~abstract~~ class, to create object of abstract class their is approach is called as "concrete class".
- A class which provides definition for all the incomplete methods (abstract methods) which are present inside abstract class with the help of extends keywords is known as "concrete class".

- example :-

If 10 incomplete methods are present in abstract class then subclass should provide defination of all the 10 abstract methods then only subclass is considered to be as concrete class, if it fails to provide defination for all the 10 methods then subclass should be declared as abstract class.

- syntax for declaring / providing defination in concrete class :-

```
public class students extends Velocity {
    public void test2() {          // provide {} - defination
        ----;                          in concrete class
    }
    public void test3() {          // provide defination
        ----;
    }
}

public class Test {
    public static void main (String [] args) {
        students d = new velocity();
        d.test();
        d.test2();
        d.test3();
    }
}
```

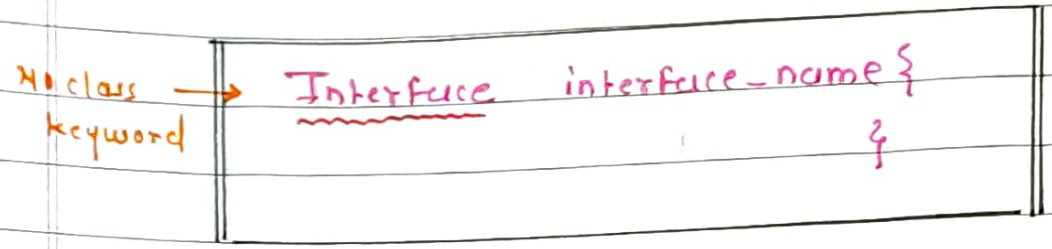
(concrete class)

(Test class)

5) Interface :-

- Interface is one of the oops principal, it is 100 % abstract in Nature.
- An Interface in java is a blueprint of class. It has a ~~static~~ Constants and ~~abstract~~ in Nature.
- The Interface in java is a mechanism to achieve abstraction.

- There can be only abstract methods in the java interface, not method body.
- It is used to achieve abstraction and multiple inheritence in Java.
- From Java 8 we can have default and static method in an interface as well as private method.
- Syntax:-



Features of Interface :-

1) Datamember declared inside interface by default static and public.
2) methods declared inside interface are default public and abstract.
3) Constructor concept is not present inside interface.
4) object of interface can't be created.
5) Interface supports multiple inheritence.

- After Java 8 we can declare static method and variable in interface.
- To create object of interface we need to make use of "Implementation class".

Implementation class :-

- A class which provides implementation for all the incomplete methods of interface, with the help of "implements" keyword is known as "Implementation class".
- At the time of implementation declared method with public access specifier Cumpulsory.

- example :-

```
public Interface demo {                          // Interface class
final & static  →  int a = 10;                   // 100% abstract method
    public void test0();                         // Interface class
    public void test1();
default & public & abstract  public void test2();
}

public class demo1 implements demo {             // Implementation class
compulsory →  public void test0() {
                ...
              }                                  // compulsory complete all methods here.
    public void test1() {
                ...
              }
    public void test2() {
                ...
              }
}
```

Encapsulation :-

- **Encapsulation** is used to hide the data, internal structure and implementation details of a object.
- All interaction with the object is through a public interface of operations.
- The user knows only about the interface, any changes to the implementation does not affect the user.
- Encapsulation in java is a mechanism to wrap up variables (data) and methods (code) together as a single unit.
- It is the process of hiding information details and protecting data and behaviour of the object.
- In Encapsulation we used getter and setter method to read and write purpose.

- For Encapsulation we just declare class as Normal class.
- providing security to the methods and Variables we are declaring private access specifier infront of get and set method.
- If we want to access those get, set method in another class for that we have to create an object.
- Example :-
  "Binfolder" and "class" is an example of Encapsulation.
- Benefits of Encapsulation :-
  1) To achieve data hiding (one of way to hide code)
  2) It provide control over the data
  3) class is easy to test.
  4) Unit level and Integration testing is used with Encapsulation
  5) Encapsulation is binding the data with its related functions. Here functionalities means "methods" and data means "Variables".
  6) So we keep Variables and methods in one place. that is "class", class is base for encapsulation.
  7) with java encapsulation, you can hide (restrict access) to critical data member in our code, which improves security.
  8) If a data member is declared "private", then it is can only be accessed within the same class.
  9) It can only be accessed using the method of their current class.

7) Generalization :-
- Generalization is one of the generic principle in all the programming.

- Extracting all the common important properties and declaring it in superclass and providing implementation according subclass specification is known as "generalization"
- generalization file can be regular java class or abstract class or interface but interface is recommended.
- In generalization we can used all the oop's concept i.e inheritence and interface concept.
- Generalization is one of the generic principle in all programming language where we use each and every concept from the oops.

* "This" and "super" keyword :-

1) "This" keyword :-

- "This" keyword is used to access global variable from the same class.
- mainly when global and local variable has same variable name.
- example :-

```
public class demo {
    public static void main (String [] args) {
        int a = 50; // global variable
    public void test () {
            int a = 40; // local variable
            system.out.println (a); // print local variable
            system.out.println (this.a); // print global variable
    }
}
```

output: 40
50

2) "super" keyword :-

- "super" keyword is used to access global variable from super class or different class.

- example :-

Superclass

```
public class demo {
    public static void main (String [] args) {
        int a = 10;
        public void test() {
            ..... }
```

Subclass

```
public class demo1 extends demo {
    int a = 20;
    public void test1() {
        System.out.println (a); // print a of this class
        System.out.println (super.a); // print 'a' of superclass
    }
}
```

output :- 20
10

★ Blocks in Java :-

- Anywhere inside the java programe " { } " represents block.
- Blocks are used to initialize data member, variable and to declare printing statements and executable statements.
- example :-

```
class {
    static () {
        // Declaration (Data member, Variable)
        // printing statement
        // execution statement
    }
    Non-static () { ← (dont write by default)
    }
}
```

Static Block

Non-static Block

- There are two types of Blocks they are as follows:
  1) static Block
  2) Non-static Block

1) static Block :-
- If programer is declaring static keyword before block then it is considered to be as "static block".
- static block can be used to initialize static data and member only.
- At the time of programe execution before the execution of main method static block is going to be executed.
- static block is going to executed only once the lifetime of the programe.
- Any number of static block can be declared in single java class.

2) Non-static Block :-
- Inside Java class if programmer is declaring a block without declaring any keyword then it is considered to be as "Non-static Block."
- Non-static blocks are used to initialize non-static data members as well as static members.
- printing statements as well as executable statements can be declared in block.
- To execute Non-static block only class call is not sufficient so to execute Non-static block object of class is mandatory.