Module – III

Object Oriented Programming

OpenEdge ABL
ABAP
Visual Basic
Scala
F#
FoxPro
Perl
Logo
Objective-C
Erlang
C#
D
ML
Transact-SQL
Prolog
ColdFusion
Clarion
JavaScript
Forth
Python
PostScript
ActionScript
PL/SQL
Q
COBOL
Assembly
Scratch
C++
Scheme
R
Ada
C
Lua
Java
Haskell
Go
Fortran
MATLAB
Io
Yacc
TCL
Ruby
PHP
Mathematica
Delphi
PL/I
Max/MSP
Pascal
cT
Dart
SAS
Haskell
Tcl
Groovy
Lisp
Swift
Scala

What you will Learn

- Class, Objects
- Method and it’s type
- Access Modifiers(private, public, default, protected)
- Non Access Modifiers(final, static, abstract)
- Inheritance & it’s types
- Polymorphism
- Encapsulation
- Constructors
- Method overriding
- Abstraction, Interface
- Packages and API
- enums

OOPs (Object-Oriented Programming System)

- **Object** means a real-world entity such as a pen, chair, table, computer, watch, etc.

- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.

- It simplifies software development and maintenance

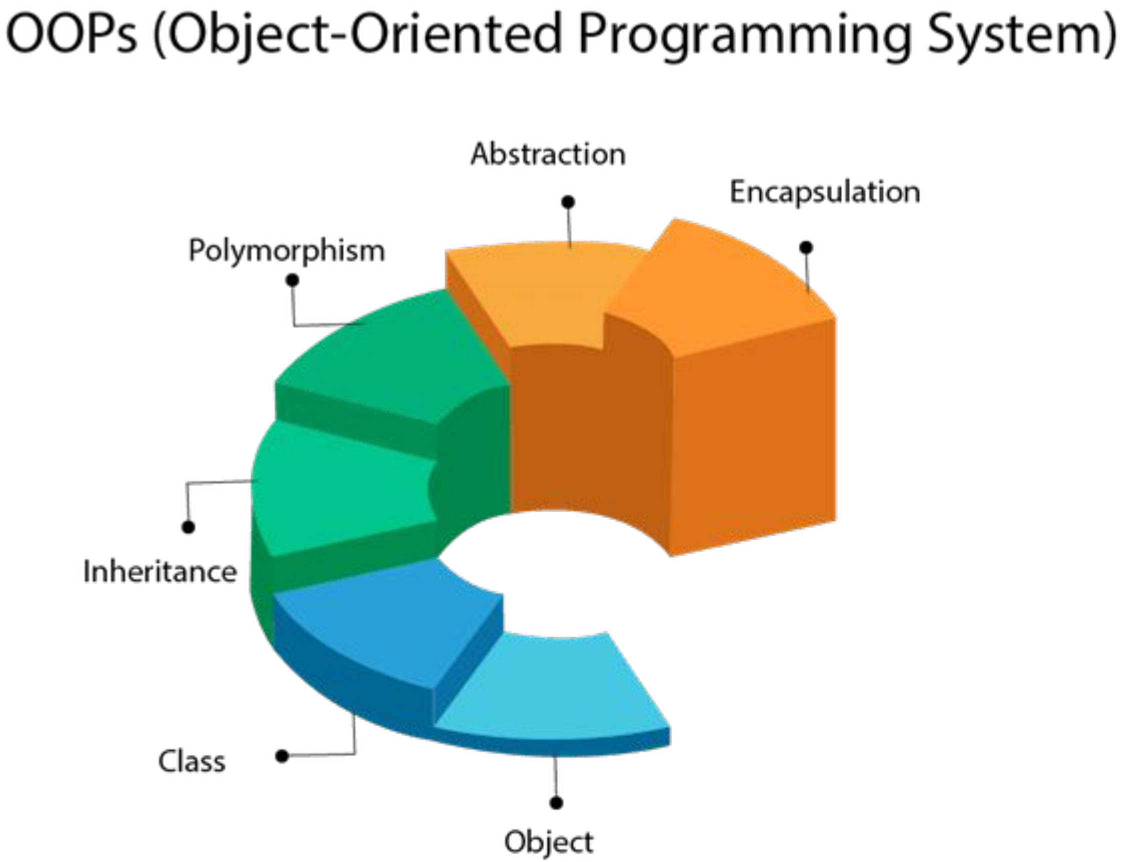
OOPs (Object-Oriented Programming System)

Concepts/Features:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

OOPs (Object-Oriented Programming System)

Main Pillars



What is Method ?

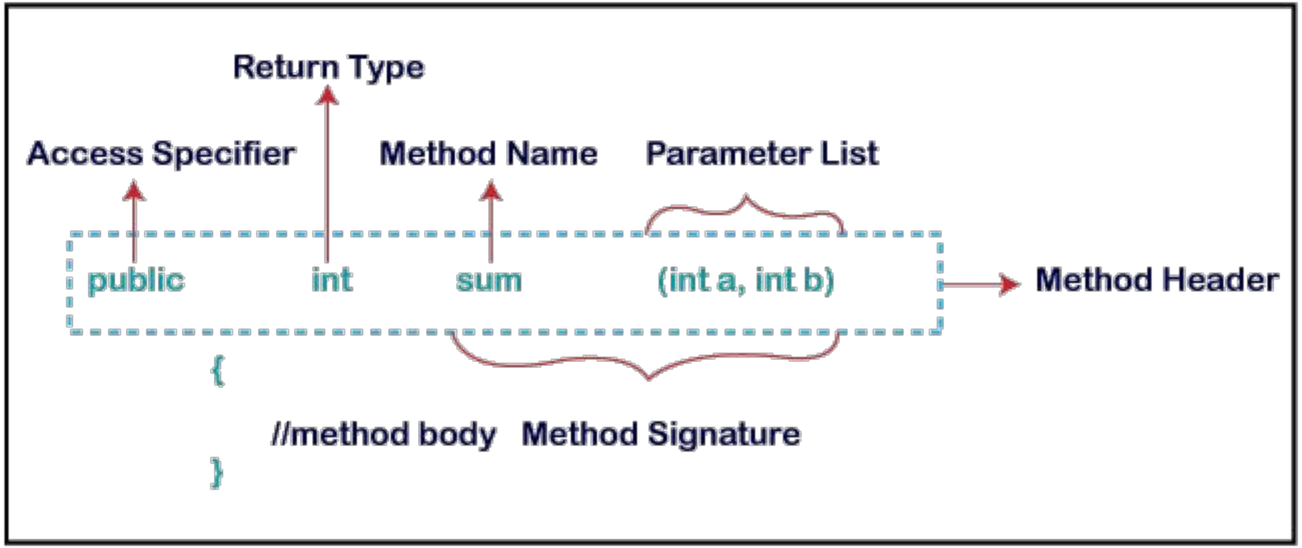
- **method in Java** is a collection of instructions that performs a specific task
- It provides the reusability of code means We define a method once and use/invoke it many times. We do not require to write code again and again
- easily modify code using **methods**
- The most important method in Java is the **main()** method

Method declaration

- method declaration provides information about method attributes, such as visibility, return-type, name, and arguments

- It has six components that are known as **method header**

Method Declaration



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Method Types

Two types of methods in Java:

Predefined Method

User-defined Method

Pre-defined Method

- predefined methods are the method that is already defined in the Java class libraries

- It is also known as the **standard library method** or **built-in method**.

- We can directly use these methods just by calling them in the program at any point.

- Some pre-defined methods are **length(), equals(), compareTo() etc**

Example

predefined.java

```
public class  predefined
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number i
s: " + Math.max(9,7));
}
}
```

Output:

```
The maximum
number is: 9
```

predefined methods used : **main(), print(),** and **max()**

User-defined Method

- The method which is defined by the user or programmer is known as **a user-defined** method.

Example

userdefined.java

```
public class  userdefined
{
public static void check(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

userdefined methods used : check

Method overloading in Java

- If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.

Method overloading in Java – Advantages

- Method overloading **increases the readability of the program**.

Ways to overload Method

Two ways to overload the method in java

- By changing number of arguments
- By changing the data type

Example

```
class Calculation{
  void sum(int a,int b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  Calculation obj=new Calculation();
  obj.sum(100,20,30);
  obj.sum(46,87);

  }
}
```

Importance of Constructor in Java

- In Java, a constructor is a block of codes similar to the method.

- It is called when an instance of the class is created.

- Every time an object is created using the new() keyword, at least one constructor is called.

Constructor in Java – Rules

Two rules defined for the constructor

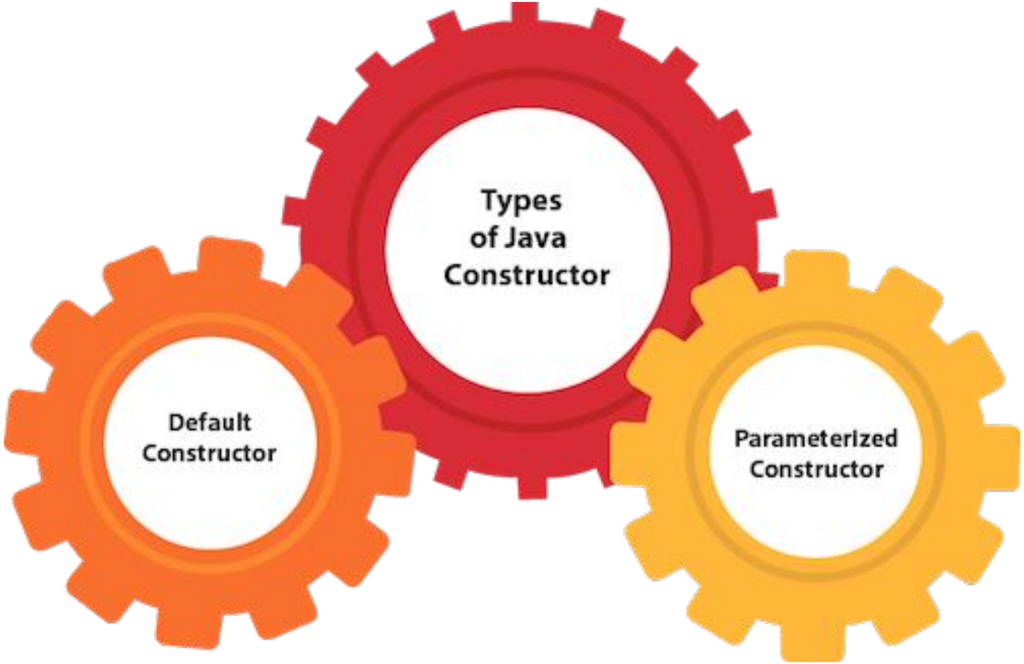
- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

Constructor in Java – Types

Two types of Constructor

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

Constructor in Java – Types



Constructor in Java – Types

Default Constructor

- constructor is called "Default Constructor" when it doesn't have any parameter

Syntax of default constructor:

```
<class_name>(){}
```

Default Constructor – Example

```
//Java Program to create and call a default constructor
class Student{
//creating a default constructor
Student(){System.out.println(“Learning Java");}
//main method
public static void main(String args[]){
//calling a default constructor
Student  s=new  Student();
}
}
```

Constructor in Java – Types

Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor

Parameterized Constructor – Example

```
class Student{
  int id;
   String name;
   //creating a parameterized constructor
   Student(int i,String n){
  id = i;
   name = n;
   }
  //method to display the values
   void display(){System.out.println(id+" "+name);}
public static void main(String args[]){
   //creating objects and passing values
   Student s1 = new Student(10,“Suraj");
   Student s2 = new Student(20,“Amit");
   //calling method to display the values of object
   s1.display();
   s2.display();
  }
}
```

Constructor overloading in Java

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists

Example

```
class Student{
         int id;
     String name;
     int age;
    Student(int i,String n){
    id = i;
    name = n;
    }
    Student(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+
name+" "+age);}
```

```
public static void main(String args[]){
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan",2
5);
    s1.display();
         s2.display();
   }
}
```

Inheritance

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

In other words, when subclass/child class acquires all the properties and behaviours from parent/super class.

Inheritance allows us to reuse of code, it improves reusability in your java application.

Inheritance in Java

Note: The biggest **advantage of Inheritance** is Code Reusability

means that the code that is already present in base class need not be rewritten in the child class

Inheritance in Java – Syntax

syntax of Java Inheritance

```
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.

Types of Inheritance in Java

Various types of inheritance in Java

- Single- Level inheritance
- Multiple Inheritance
- Multi-Level Inheritance
- Hierarchical Inheritance

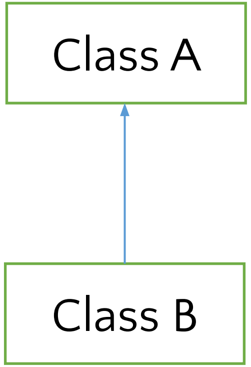
SINGLE INHERITANCE

Single Inheritance

When a class inherits another class, it is known as a *single inheritance*.

SINGLE INHERITANCE – Flowchart

- Here, Class A is your parent/super class and Class B is your sub/child class which inherits the properties from parent class.



Example

```
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();
}
```

Output:

```
barking
eating
```

MULTILEVEL INHERITANCE

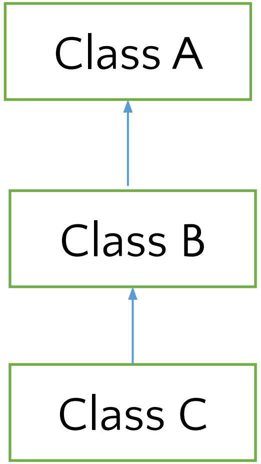
When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.

Chain of Inheritance

MULTILEVEL INHERITANCE – Flowchart

- If we talk about the flowchart, class B inherits the properties and behavior of class A and class C inherits the properties of class B

- Here A is the parent class for B and class B is the parent class for C



Example

```
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();
}}
```

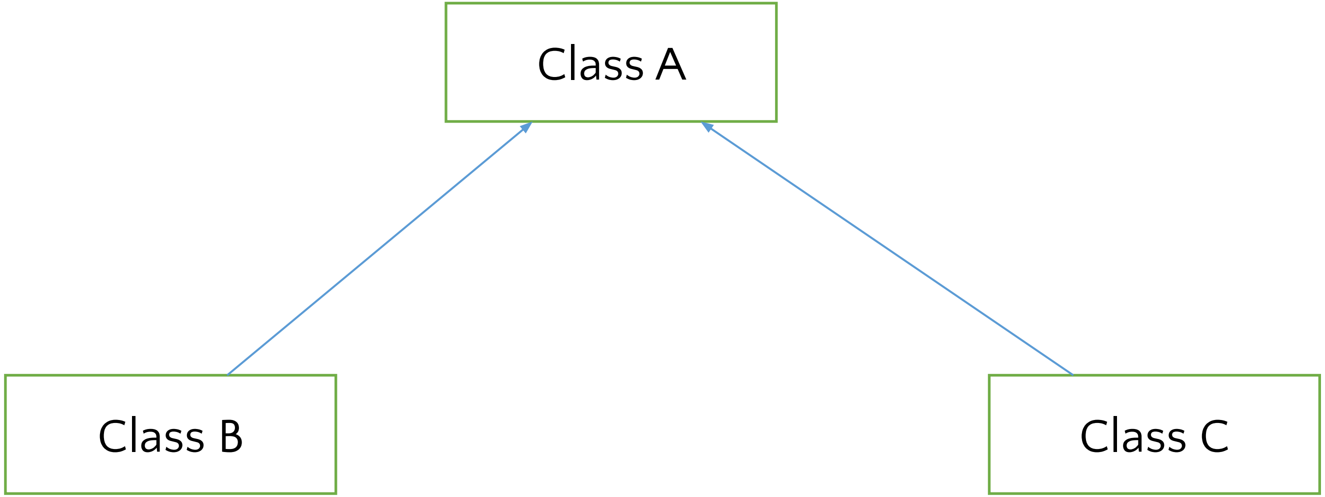
Output:

```
weeping...
barking...
 eating...
```

HIRERARCHICAL INHERITANCE

- When a class has more than one child classes (subclasses) or in other words, more than one child classes have the same parent class, then such kind of inheritance is known as hierarchical

HIRERARCHICAL INHERITANCE – Flowchart



Example

```
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();
}
}
```

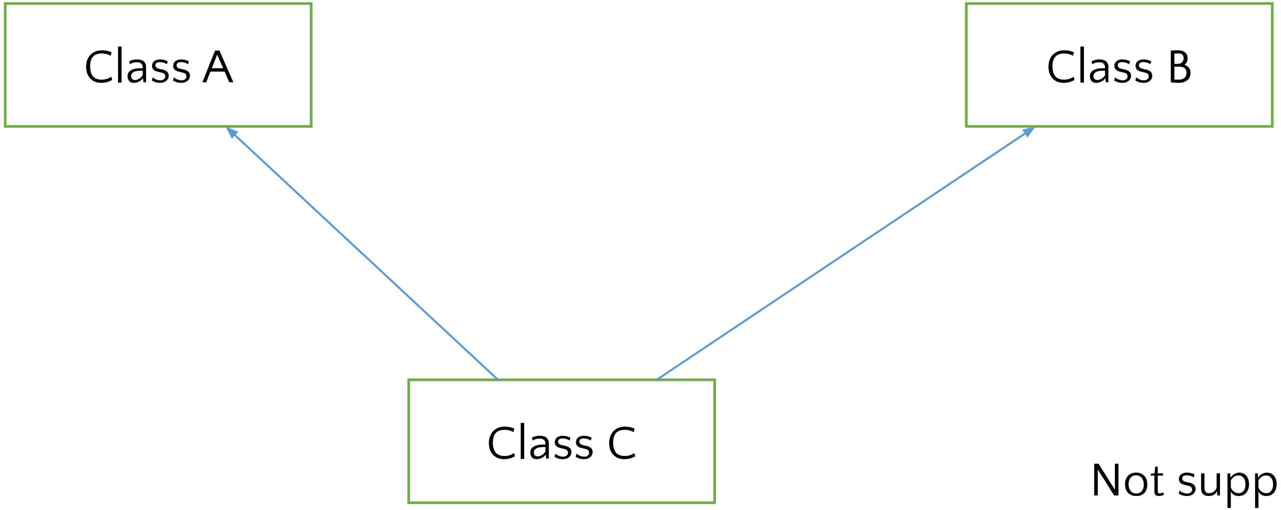
Output:

```
meowing...
eating...
```

Why MULTIPLE INHERITANCE is not supported?

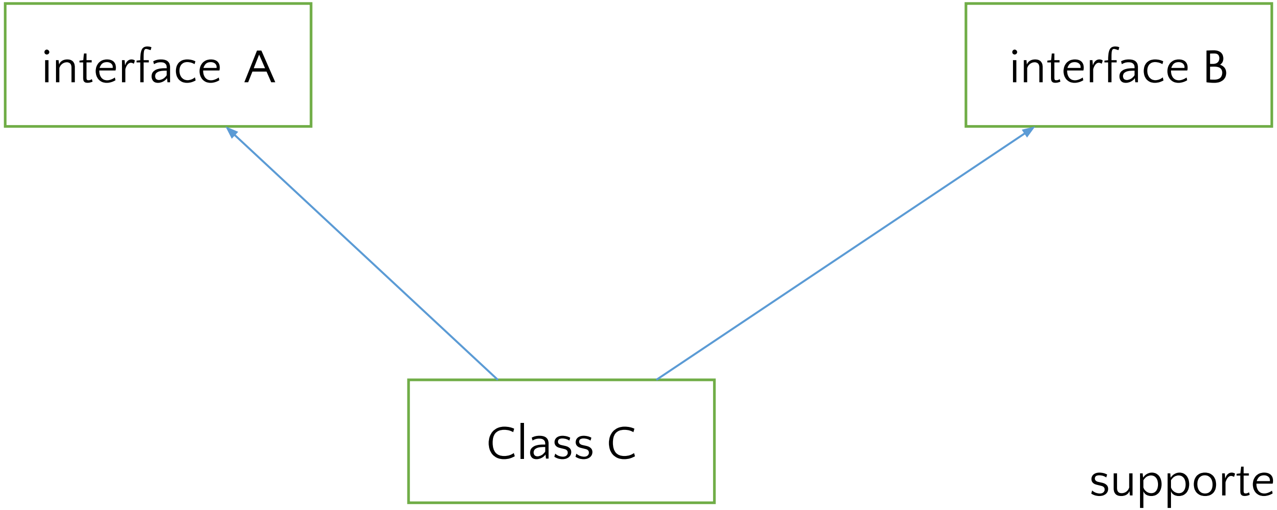
- To reduce the complexity and simplify the language, multiple inheritance is not supported in java
- Consider a scenario where A, B, and C are three classes
- The C class inherits A and B classes
- If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class

MULTIPLE INHERITANCE – Flowchart



Not supported through class

MULTIPLE INHERITANCE – Flowchart



supported
through interface

Example

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{
public static void main(String args[]){
C obj=new C();
obj.msg();
}
}
```

Output:

Compile time error

Example –solved by interface

```
interface A{
void msg();
}
interface B{
void msg();
}
public class multipleinheritancedemo
implements A,B{
public void msg(){
    System.out.println("Hello Java!");
}
```

```
public static void main(String
args[]){
    multipleinheritancedemo
obj=new
multipleinheritancedemo();
    obj.msg();
}
}
```

Output:

```
Hello Java!
```

Rules for inheritance

Multiple Inheritance is NOT permitted in Java

Cyclic Inheritance is NOT permitted in Java

Private members do NOT get inherited

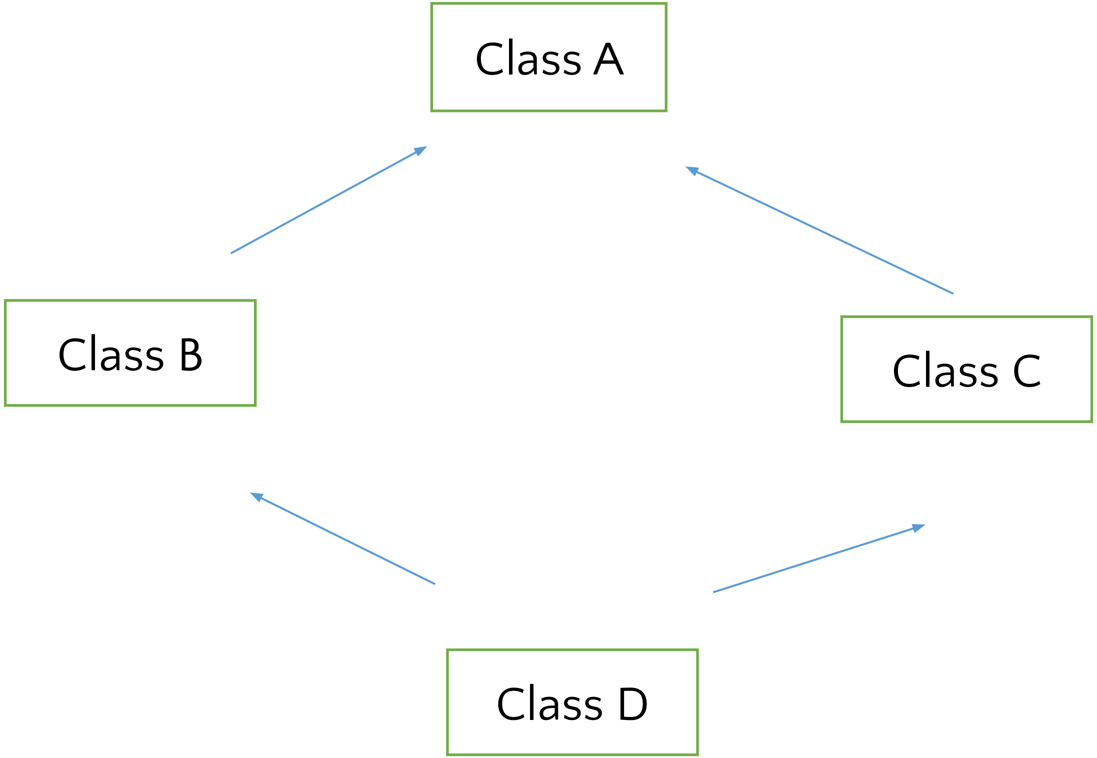
Constructors cannot be Inherited in Java

In Java, we assign parent reference to child objects

HYBRID INHERITANCE

- Combination of Hierarchical & Multiple Inheritance is called as Hybrid Inheritance.
- **Note:** Hybrid Inheritance also not supported in Java through class.
- It's possible through interface.

HYBRID INHERITANCE – Flowchart



Example

```
interface A{
void msg();
}
interface B extends A{
void disp();
}

interface C extends A{
void print();
}
public class D implements B,C{
public void msg(){
     System.out.println("Hello Java!");
}
```

```
public void print(){
     System.out.println("Inheritance");
}
  public void disp(){
      System.out.println("Let's do it");
  }
    public static void main(String args[]){
     D obj=new D();
     obj.msg();
     obj.disp();
     obj.print();
}
}
```

Output:

```
Hello Java!
Let's do it
Inheritance
```

Method Overriding in Java

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

- Method overriding is used for runtime polymorphism

Method Overriding in Java

Rules for Java Method Overriding

The method must have the same name as in the parent class

The method must have the same parameter as in the parent class

The method must have the same return type as in the parent class

There must be an IS-A relationship (inheritance).

Example

```
class ICT{
  //defining a method
  void  training(){System.out.println(“ICT is
here ");}
}
//Creating a child class
class Student extends ICT{
  //defining the same method as in the parent
class
  void training(){System.out.println("Java
training");}
```

```
public static void main(String
args[]){
  Student obj = new
Student();//creating object
  obj.training();//calling method
  }
}
```

Output:

```
Java training
```

this keyword in Java

- In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

- this keyword can be used to refer current class instance variable.
- this() can be used to invoke current class constructor.
- this keyword can be used to invoke current class method (implicitly)

super keyword in Java

- 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 super keyword in Java

- 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

Example

- super Is Used To Refer Immediate Parent Class Instance Variable

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);
System.out.println(super.color);
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}
}
```

Example

- super can be used to invoke parent class method

```
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();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}
}
```

Example

- super is used to invoke parent class constructor

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}
}
```

Non-Access Modifier in Java

- Non-access modifiers are those keywords that do not have anything related to the level of access but they provide a special functionality when specified.

Non-Access Modifier in Java

- **Final**
  Final keyword can be used with variable, method or class. It prevents from its content from being modified. When declared with class, it prevents the class from being extended.
- **Static**
  The static modifier is used with class variables and methods which can be accessed without an instance of a class. Static variables have only single storage. All objects share the single storage of static variable. They can be accessed directly without any object.
- **Abstract**
  abstract can be used with class and methods. An abstract class can never be instantiated and its purpose is only to be extended.

final keyword in Java

- The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:
- variable
- method
- class

final keyword in Java

Java final variable

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

Java final method

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

Java final class

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

static keyword in Java

- The **static keyword** in Java is used for memory management mainly
- We can apply static keyword with variables, methods, blocks and nested classes
- The static keyword belongs to the class than an instance of the class

static keyword in Java – Advantages

Advantages of static variable

- It makes your program **memory efficient** (i.e., it saves memory).

static keyword in Java

Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

- The static variable gets memory only once in the class area at the time of class loading.

static keyword in Java

Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Java static block

If you apply static keyword with any method, it is known as static method.

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Encapsulation

Encapsulation in Java

- Process of wrapping code and data together into a single unit

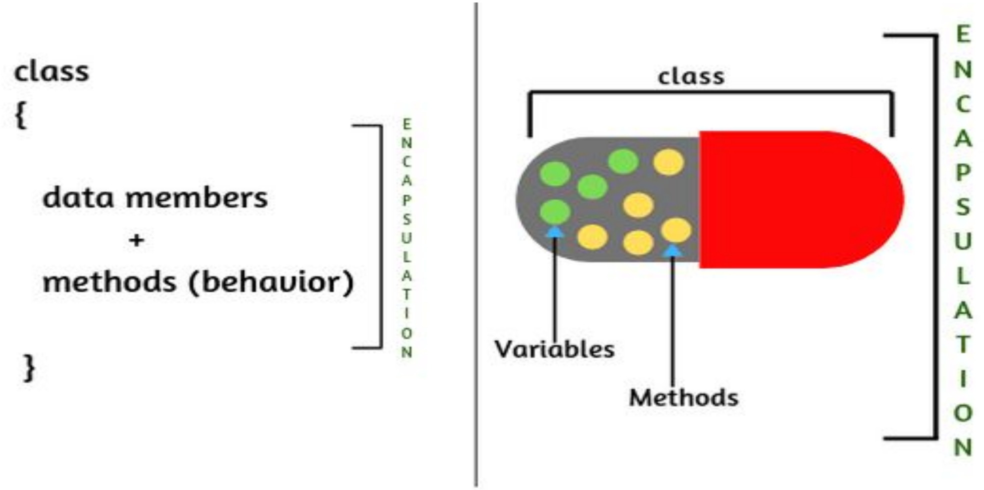


Fig: Encapsulation

Why Encapsulation?

- Better control of class attributes and methods
- Class attributes can be made read-only, or write-only
- Flexible
- Data security

How to do Encapsulation?

- Declare class variables/attributes as private

- provide public **get** and **set** methods to access and update the value of a private variable

Example

- The **get** method returns the variable value, and the **set** method sets the value

```
class Subclass{
 private String name;
 public String getName(){
  return name;
 }
 public void setName(String n){
   name = n;
 }
}
```

```
class Main{
 public static void main(String rgs[]){
  Subclass S1 = new Subclass();
  String getname = S1.getName();
  System.out.println(getname);
  S1.setName("Codemithra");
  System.out.println(S1.getName());
 }
}
```

Read only class

- The **get** method returns the variable value, and the **set** method sets the value

```
class Subclass{
  private String name;
  public String getName(){
    return name;
  }
  public void setName(String n){
     name = n;
  }
}
```

```
class Main{
  public static void main(String rgs[]){
    Subclass S1 = new Subclass();
    String getname = S1.getName();
    System.out.println(getname);
    S1.setName(“ICT Academy");
    System.out.println(S1.getName());
  }
}
```

Access Modifiers
and
Package

Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.

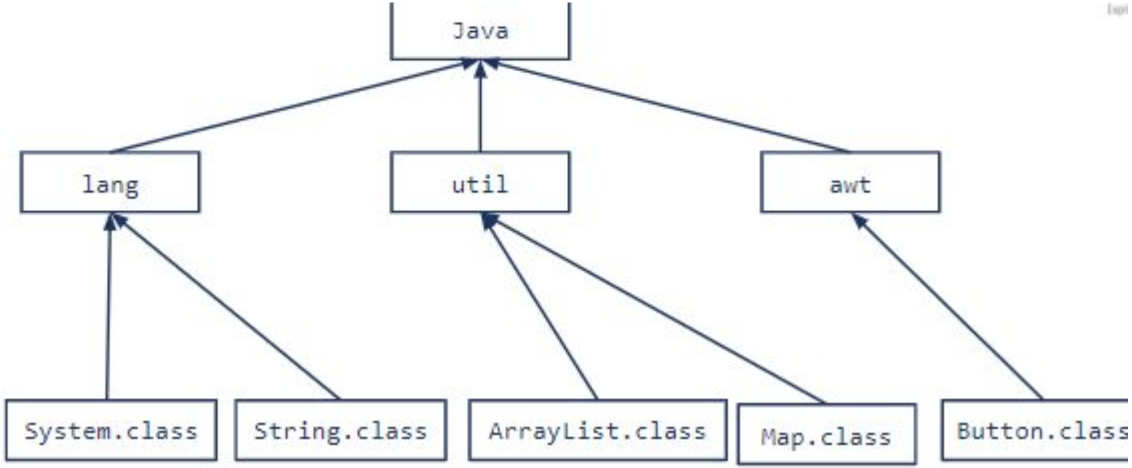
Different Types of packages:

- Built-in package.
- User-defined package.

Advantages of Packages

- Easily maintained
- Java package provides access protection
- Java package removes naming collision

Built in Packages



Creating Package

- The **package keyword** is used to create a package

```
package mypack;
public class main{
public static void main(String args[]){
  System.out.println("Welcome to package");
}
}
```

Access Package from another Package

- The different ways to access the package from outside the package

- import package.*;

- import package.classname;

import package.* – Example

```
package pack;

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

```
import pack.*;

class Main{
  public static void main(String args[]){
    Subclass obj = new Subclass();
    obj.msg();
  }
}
```

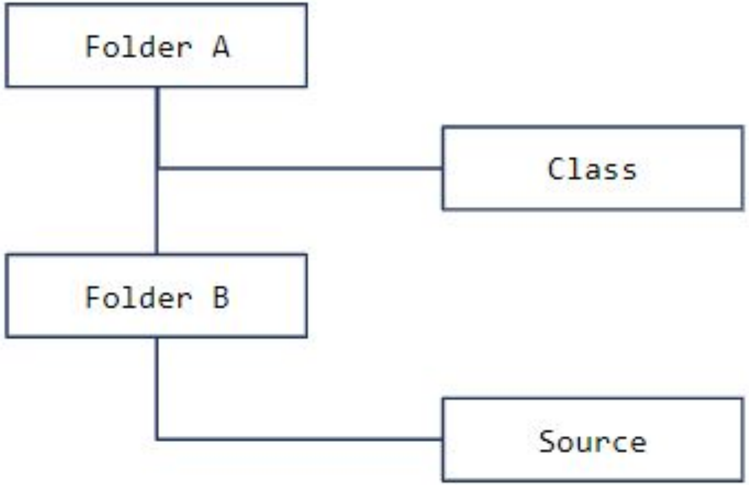
package.classname – Example

```
import pack.Subclass;

class Main{
  public static void main(String args[]){
    Subclass obj = new Subclass();
    obj.msg();
  }
}
```

```
package pack;

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

Subpackage

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



Access modifier

- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.

- We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

Types of Access modifier

- **private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

- **default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Types of Access modifier

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

- **public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package

Access modifier – Chart

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

private access modifier – Example

```
class Subclass{
  private int data = 40;
  private void msg(){
    System.out.println("Hello");
  }
}
```

```
public class Main{
  public static void main(String args[]){
    Subclass obj = new Subclass();
    System.out.println(obj.data);    ✗
    obj.msg();          ✗
  }
}
```

Access Modifier	within class	within package	outside package by subclass only	outside package
private	Y	N	N	N

default access modifier – Example

- If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package

```
package pack;
class Subclass{
 void msg(){
   System.out.println("Hello");
 }
}
```

```
import pack.*;
class Main{
 public static void main(String args[]){
  Subclass obj = new Subclass();   ✗
  obj.msg();   ✗
 }
}
```

Access Modifier	within class	within package	outside package by subclass only	outside package
default	Y	Y	N	N

protected access modifier – Example

- The **protected access modifier** is accessible within package and outside the package but through inheritance only

protected access modifier – Example

```
class Subclass{
  protected void msg(){
    System.out.println("Hello");
  }
}
class Main extends Subclass{
  public static void main(String args[]){
    Subclass obj = new Subclass();
    obj.msg();
  }
}
```

Access Modifier	within class	within package	outside package by subclass only	outside package
protected	Y	Y	Y	N

public access modifier – Example

- The **public access modifier** is accessible everywhere

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

```
public class Main{
  public static void main(String args[]){
    Subclass obj = new Subclass();
    System.out.println(obj.data);
    obj.msg();
  }
}
```

Access Modifier	within class	within package	outside package by subclass only	outside package
public	Y	Y	Y	Y

Concept of Polymorphism

Polymorphism

- **Polymorphism in java** is a concept by which we can perform a *single action by different ways*,or

- When **one task is performed by different ways** i.e. known as polymorphism. For example: to convense the customer differently, to draw something e.g. shape or rectangle etc.

- 2 Greek words: **poly** and **morphs**. The word **"poly"** means many and **"morphs**" means forms

Types of Polymorphism

There are two types of polymorphism :

- Static/compile-time polymorphism
- Dynamic/runtime polymorphism

Compile time Polymorphism

- If you overload a static method in Java, it is the example of compile(Static) time polymorphism

- Method overloading is a perfect example of compile-time polymorphism

Example

```
class Subclass{
  int add(int a, int b){
    return a+b;
  }
  int  add(int a, int b, int c){
    return a+b+c;
  }
}
```

```
public class Main{
  public static void main(String args[]){
    Subclass obj = new Subclass();
    System.out.println(obj.add(10, 20));
    System.out.println(obj.add(10,
20,30));
  }
}
```

Runtime Polymorphism

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time

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

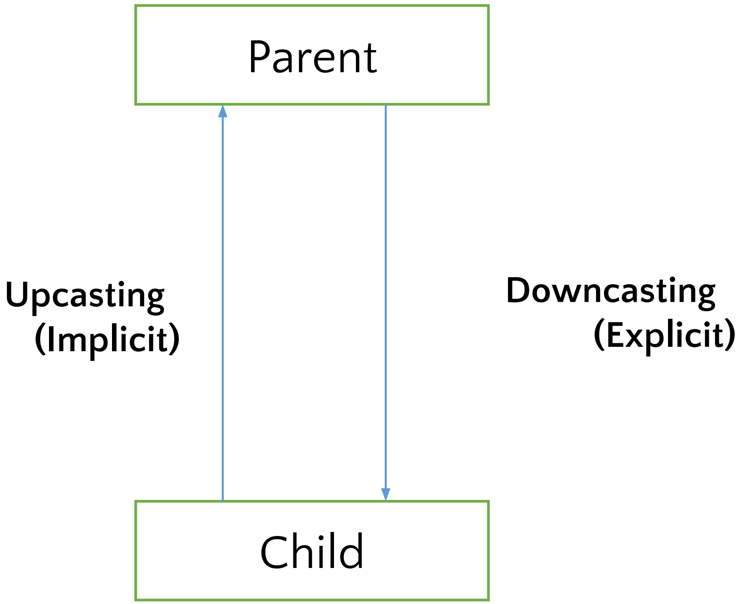
- Method overriding is a perfect example of runtime polymorphism

Example

```
class Bike{
  void run(){
    System.out.println("running");
  }
}

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

Upcasting and Downcasting

Parent p = new Child();



Upcasting – Example

```
class Parent{
   void method(){
    System.out.println("Method from Parent");
   }
}
class Child extends Parent{
   void method(){
     System.out.println("Method from Child");
   }
}
```

```
public class Main{
 public static void main(String[] args){
        Parent p = new Child();
        p.method();
   }
}
```

Differences

Compile Time Polymorphism	Runtime Polymorphism
• It provides fast execution because the method that needs to be executed is known early at the compile time	• It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime
• Compile time polymorphism is less flexible as all things execute at compile time	• Run time polymorphism is more flexible as all things execute at run time

Abstraction(abstract class, interface)

Abstraction

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

- Abstraction can be achieved with either abstract classes or interfaces

- Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details

Abstraction – Real Life Example

- sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery

- ATM Machine

Abstraction – Ways

Two ways to achieve abstraction in java

- Abstract class (0 to 100%)
- Interface (100%)

abstract class

- An abstract class must be declared with an abstract keyword
- It can have abstract and non-abstract methods
- It cannot be instantiated
- It can have constructors and static methods also
- Abstract classes cannot be used to instantiate objects

Abstraction

- Abstract Class

```
abstract class Class_name{}
```

- Abstract Method

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

```
abstract void Method_name();//no method body and abstract
```

abstract keyword

- abstract is a non-access modifier in java applicable for classes, methods but **not** variables
- It is used to achieve abstraction which is one of the pillar of Object Oriented Programming(OOP)

abstract method rules

- Any class that contains one or more abstract methods must also be declared abstract

- The following are various **illegal combinations** of other modifiers for methods with respect to *abstract* modifier :

- final
- abstract native
- abstract synchronized
- abstract static
- abstract private
-

Example

```
abstract class Subclass{
  abstract void run();
}
class Main extends Subclass{
  void run(){
    System.out.println("Hi");
  }
  public static void main(String args[]){
    Subclass obj = new Main();
    obj.run();
  }
}
```

Example

```
abstract class Shape{
  abstract void draw();
}
class Rectangle extends Shape{
  void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
  void draw(){System.out.println("drawing circle");}
}
class Main{
  public static void main(String args[]){
    Shape s=new Circle1();
    s.draw();   }
}
```

- Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes

- Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the factory method

- In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked

Constructor, Data member and Methods

```
abstract class Subclass{
  Subclass(){
   System.out.println("Subclass..");
   }
  abstract void Hey();
  void Hi(){
   System.out.println("Hi Method");
   }
}
```

```
class Subclass2 extends Subclass{
  void Hey(){
   System.out.println("Subclass2..");
   }
}
class Main{
  public static void main(String args[]){
    Subclass obj = new Subclass2();
    obj.Hey();
    obj.Hi();
  }
}
```

Encapsulation vs Abstraction

- Encapsulation is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding)

- While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation

interface

interface

- An **interface in Java** is a blueprint of a class
- Has static constants and abstract methods
- Mechanism to achieve abstraction
- It is used to achieve abstraction and multiple inheritance in Java

Why interface

- It is used to achieve **abstraction**
- By interface, we can **support** the functionality of **multiple inheritance**
- It can be used to achieve **loose coupling**

interface

- R**epresents the IS-A relationship**
- It cannot be instantiated just like the abstract class

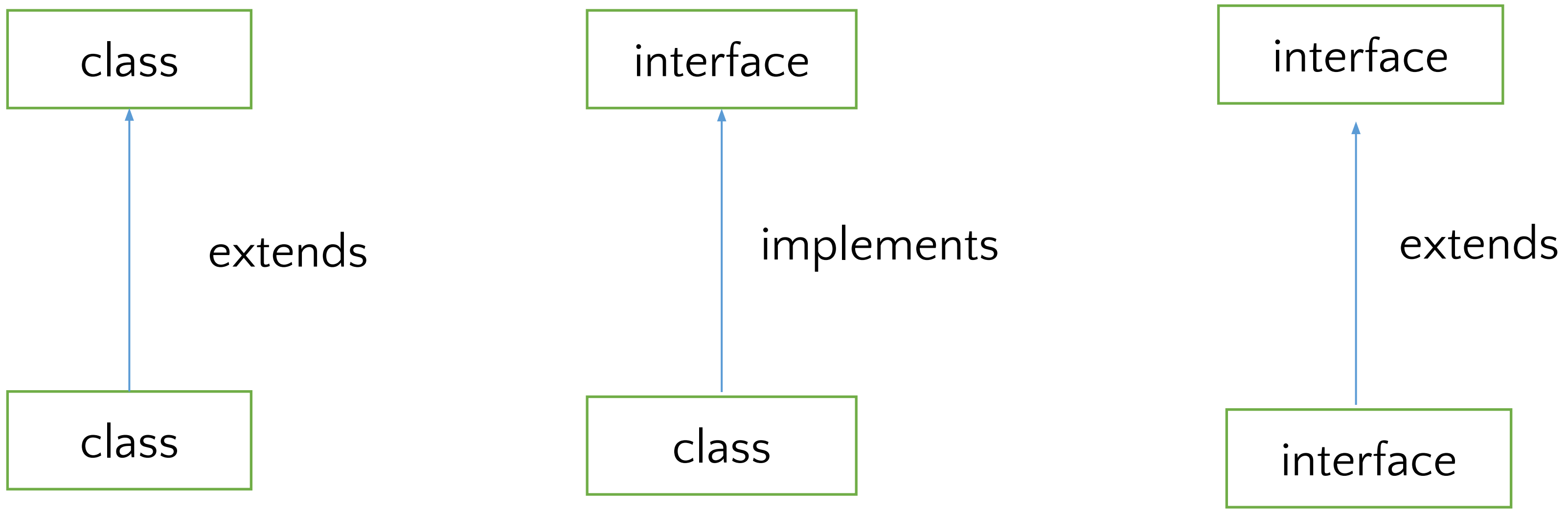
Interface – Declaration

- An interface is declared by using the **interface** keyword
- All the fields are **public**, **static** and **final** by default
- A class that implements an interface must implement all the methods declared in the interface

Interface – Syntax

```
interface <interface_name>{
// declare constant fields
// declare methods as abstract
}
```

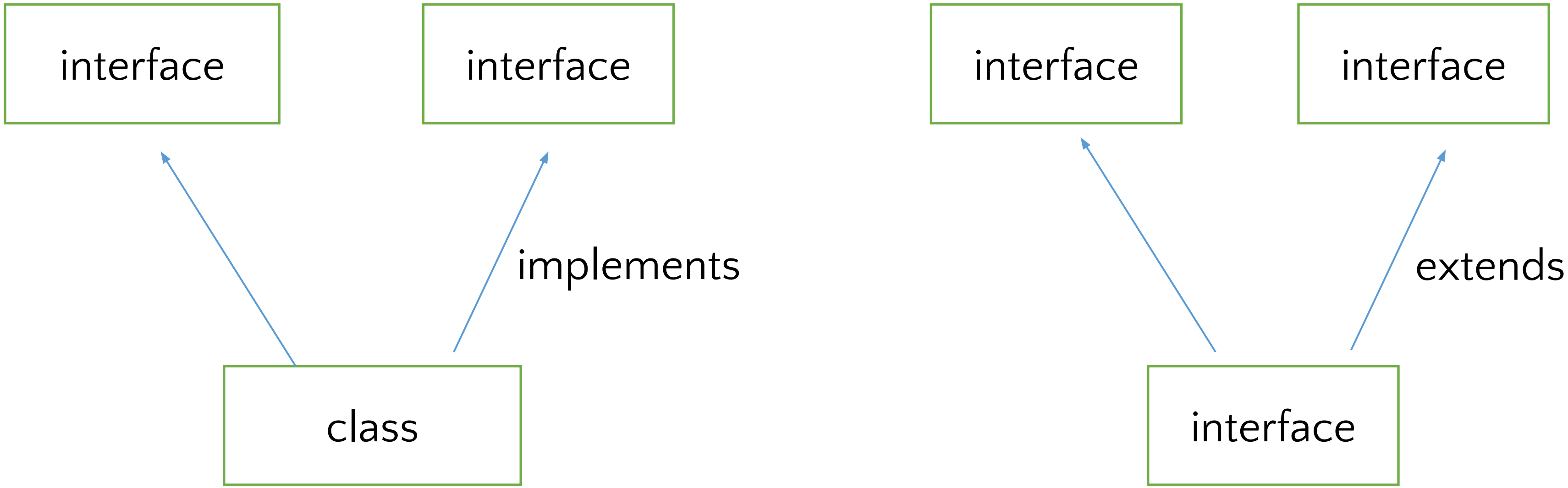
Relationship between Class and Interface



Example

```
interface Subclass{
  void print();
}
class Main implements Subclass{
  public void print(){
    System.out.println("Hello");
  }
  public static void main(String args[]){
    Main obj = new Main();
    obj.print();
  }
}
```

Multiple Inheritance



interface

interface

implements

class

interface

interface

extends

interface

Example

```
interface interface1{
  void Submeth1();
}
interface interface2{
  void Submeth2();
}
```

```
class Main implements interface1,interface2{
  public void Submeth1(){
    System.out.println("Hello");
  }
  public void Submeth2(){
    System.out.println("Welcome");
  }
  public static void main(String args[]){
    Main obj = new Main();
    obj.Submeth1();
    obj.Submeth2();
  }
}
```

Example

```
interface interface1{
 void Submeth();
}
interface interface2{
 void Submeth();
}
class Main implements interface1,interface2{
 public void Submeth(){
  System.out.println("Hello");
 }
 public static void main(String args[]){
  Main obj = new Main();
  obj.Submeth();
 }
}
```

Interface inheritance

```
interface interface1{
  void Submeth1();
}

interface interface2 extends interface1{
  void Submeth2();
}
```

```
class Main implements interface1{
  public void Submeth1(){
    System.out.println("Hello");
  }
  public void Submeth2(){
    System.out.println("Welcome");
  }
  public static void main(String args[]){
    Main obj = new Main();
    obj.Submeth1();
    obj.Submeth2();
  }
}
```

Java 8 Interface – default Method

```
interface interface1{
  void Submath1();
  default void Defmath1(){
    System.out.println("default method");
  }
}
class interface2 implements interface1{
  public void Submath1(){
    System.out.println("interface2 method..");
  }
}
```

```
class Main{
  public static void main(String arg[]){
    interface1 d=new interface2();
    d.Submath1();
    d.Defmath1();
  }
}
```

Java 8 Interface – static Method

```
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 Main{
  public static void main(String args[]){
    Drawable d = new Rectangle();
    d.draw();
    System.out.println(Drawable.cube(3));
  }
}
```

Nested interface

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class

- Nested interfaces are declared static implicitly

Syntax

```
interface interface_name{
 ...
 interface nested_interface_name{
   ...
 }
}
```

Example

```
interface Showable{
  void show();
  interface Message{
      void msg();
  }
}
```

```
class Main implements Showable.Message{
 public void msg(){
  System.out.println("Hello nested interface");
 }
 public static void main(String args[]){
  Showable.Message message = new Main();
  message.msg();
 }
}
```

Nested interface within class

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Example

```
class Subclass{
 interface Message{
  void msg();
 }
}
class Main implements Subclass.Message{
 public void msg(){
  System.out.println("Hello nested interface");
 }
 public static void main(String args[]){
  Subclass.Message message = new Main();
  message.msg();
 }
}
```

Java enums

- The **Enum in Java** is a data type which contains a fixed set of constants
- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change)

Java enums

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Example

```
class Enum1{
//defining the enum inside the class
public enum Students { Aman, Rohan, Pankaj, Anand }
//main method
public static void main(String[] args) {
//traversing the enum
for (Students  s :  Students.values())
System.out.println(s);
}}
```