OBJECT ORIENTED PROGRAMMING CONCEPTS IN JAVA

SIMPLIFIED NOTES FOR BEGINNERS

COMPILED BY: SHER KHAN BALOCH

FULL STACK .NET DEVELOPER

Contents

Introduction

- OOP is a structure or syntax of writing a programming.
- Small Talk (1st language of OOP), Java, C#, C++, Python.
- OOP is a programming paradigm or Methodology.

Types of Programming Paradigm

Object Oriented Paradigm

Functional Paradigm

Structured Paradigm

Procedural Paradigm

Logical Paradigm

06 Main Pillars of OOP

1. Class
2. Objects and Methods
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Classes, Methods, Objects

- **Class:**
  - Class is the collection of objects.
  - Class is not the real world entity, it is just a template or blue print or prototype.
  - Class does not occupy memory.
  - Class is a Non-Primitive or reference Data type.

  Syntax:

```
Access-modifier        class              ClassName
{
            ▪  Methods
            ▪  Constructors
            ▪  Fields or Variables
            ▪  Blocks
            ▪  Nested Class
}
```

- **Methods:**
  - A set of code with performs a particular task.
  - Advantages of Methods are Code Re-usability and Code Optimization.

  Syntax:

```
Access-modifier        return-type              methodName (arguments)
{
        // Code Here.
}
```

- **Object:**
  - Object is an instance of a Class.
  - Object is a real world entity.
  - Object occupy memory.

  Object Consists on:
  1. Identity = name
  2. State or Attributes = Variables = color, age
  3. Behavior = Methods = eat, run, walk

- How to Create an Object:
  1. Using New Keyword
  2. New instance method
  3. Clone method
  4. Deserialization
  5. Factory methods

- Class: **Animal**
  Object: Dog, Cat
    - Attributes: name, age, type
    - Method: Run(), Eat(), Sleep()

- Class: **Birds**
  Object: Sparrow, Peacock
    - Attributes: name, age, color
    - Method: Fly(), Eat(), Sleep()

- Class: **Vehicle**
  Object: Car, Jeep
    - Attributes: name, model, company
    - Method: Drive(), Fuel(), Start()

1. Creating Object Using New Keyword (3 Steps)
    1. Declaration: Animal **dog**
    2. Instantiation: dog = **new**
    3. Initialization: dog = new Animal();

    - Declaration:
      It is declaring a variable name with an object type.

      Animal dog

    - Instantiation:
        - This is when memory is allocated for an object.
        - The **New** keyword is used to create the object.
        - A reference to the object that was created is returned from the new keyword.

          dog = new

    - Initialization:
        - The **New** keyword is followed by a call to a constructor. This call initializes the new object.
        - In this, the values are put into the memory that was allocated.

          dog = new Animal();

- We can use dot (.) for calling the methods of object.
  dog.run();
- If we don't write access modifiers name then it will take "Default" access modifier by default.

Example 01:

```
Class Animal
{
        Public void eat()
        {
                System.out.println(“Animal is eating”);
        }

        Public static void main (String[] agrs)
        {
                Animal dog = new Animal();
                dog.eat();
        }
}
```

Example 02:

```
Class Birds
{
        Public void fly()
        {
                System.out.println(“Bird is flying”);
        }

        Public static void main (String[] agrs)
        {
                Birds sparrow = new Birds();
                sparrow.fly();
        }
}
```

- Initialization of Objects
    1. By Reference Variables
    2. By Using Methods
    3. By Using Constructors

1. Initialize Object By **Reference Variable**

```
Class Animal
{
        String color;
        Int age;

        Public static void main (String[] args)
        {
                Animal dog = new Animal();
                dog.color = “Black”;
                dog.age = 10;

                System.out.println(dog.color + “ ” + dog.age);
        }
}
```

2. Initialize Object By **Using Method**

```
Class Animal
{
        String color;
        Int age;

        Public void setData(String c, int a)
        {
                color = c;
                age = a;
        }

        Public void getData()
        {
                System.out.println(color + “ ” + age);
        }

        Public static void main (String[] args)
        {
                Animal dog = new Animal();
                dog.setData( “Black”, 10);
                dog.getData();
        }
}
```

Constructor

- Constructor is a block similar to method having same name as that of class name.
- Constructor does not have any return type, not even void.
- The access modifiers applicable for constructor are Public, Protected, Default and Private.
- It executes automatically when we create an object.
- Constructor is used to initialize the object. It does not create object.

Example:

```
Class Test
{
          Public Test()
          {
                    System.out.println("Sher Khan Baloch");
          }

          Public static void main (String[] args)
          {
                    Test t = new Test();    ✓
                    t.Test();               ✕
          }
}
```

- **Before Constructor:**
  Example 01

```
Class Employee
{
          Int empID = 101;
          String name = "Sher Khan Baloch";

          Public static void main (String[] args)
          {
                    Employee e1 = new Employee();
                    Employee e2 = new Employee();
          }
}
```

- It is not a good approach.
- When we will create an object it will store same value again and again in memory. So what if we want to store different value? Using this approach we can't do that.
- This is not a suitable way because when we create an object it automatically add values to memory.

Example 02

```
Class Employee
{
        Int empID;
        String name;

        Public static void main (String[] args)
        {
                Employee e1 = new Employee();
                e1.empID = 101;                                  ← Extra Lines
                e1.name = "Sher Khan Baloch";

                Employee e2 = new Employee();
                e1.empID = 102;                                  ← Extra Lines
                e1.name = "Afaque Buledi";
        }
}
```

- This is also not a good approach.
- This method can be used but in this we are writing two extra lines so what if we want to create one thousand object so then we have to write two thousand extra lines.

➢ **Using Constructor**

```
Class Employee
{
        Int empID;
        String name;

        Public Employee (int empID, String name)
        {
                this.empID = empID;
                this.name = name;
        }

        Public static void main (String[] args)
        {
                Employee e1 = new Employee(101, "Sher Khan Baloch");
                Employee e2 = new Employee(102, "Afaque Buledi");
        }
}
```

Types of Constructors

1. **Default Constructor (No Argument Constructor)**
   - It is created by compiler, when we create an object.
   - Compiler will not create a constructor by itself if user has already created it.

   Example:

```
Class Test
{
        Public Test()
        {
                Super();
        }

        Public static void main (String[] args)
        {
                Test t = new Test();
        }
}
```

2. **User Defined Constructor (No Argument Constructor)**
   1. It is created by a User or Programmer but with no arguments.

```
Class Test
{
        Public Test()
        {
                // Code Here
        }

        Public static void main (String[] args)
        {
                Test t = new Test();
        }
}
```

3. **Parametrized Constructor (With Argument)**
   2. It is created by User or Programmer.

```
Class Test
{
        Public Test(String name)
        {
                // Code Here
        }

        Public static void main (String[] args)
        {
                Test t = new Test("Sher Khan Baloch");
        }
}
```

Question: Why a Constructor has not any return type?

- If compiler create by itself then which return type it assign to constructor because it depends on user to that which return type he want to use.
- Because it's main work is to initialize an object.

Inheritance (IS-A)

- It is inheriting the properties of parent class into child class.
- Inheritance is the procedure by which one object acquires all the properties and behaviors of a parent object.
- Inheritance is achieved by using **extends** keyword.
- Inheritance creates **IS-A** relationship.
- Inheritance is also known as **IS-A** relationship.

Example:

```
// Parent Class or Super Class
Class Animal
{
        Public void eat()
        {
                System.out.println("Animal is Eating");
        }
}

// Child Class or Sub Class
Class Dog extends Animal
{
        Public static void main (String[] args)
        {
                Dog d = new Dog();
                d.eat();
        }
}
```

- Dog **IS-A** Animal.

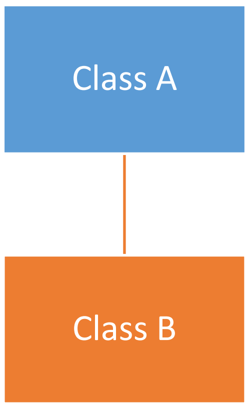
- Sparrow **IS-A** Bird.
- Car **IS-A** Vehicle.
- Surgeon **IS-A** Doctor.

1. **Uses of Inheritance**
   - Code Reusability
   - We can achieve Polymorphism (Method overriding) using inheritance.

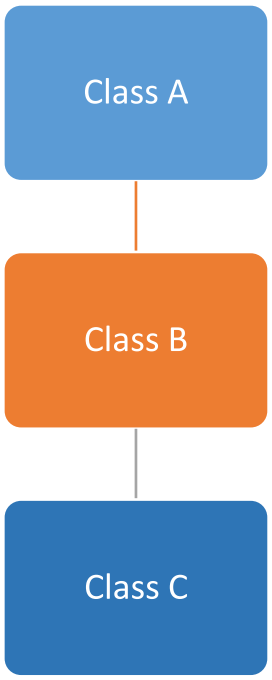
2. **Disadvantages of Inheritance**
   - Classes are tightly coupled. If we change in one class so other related classes will be effected.

Types of Inheritance

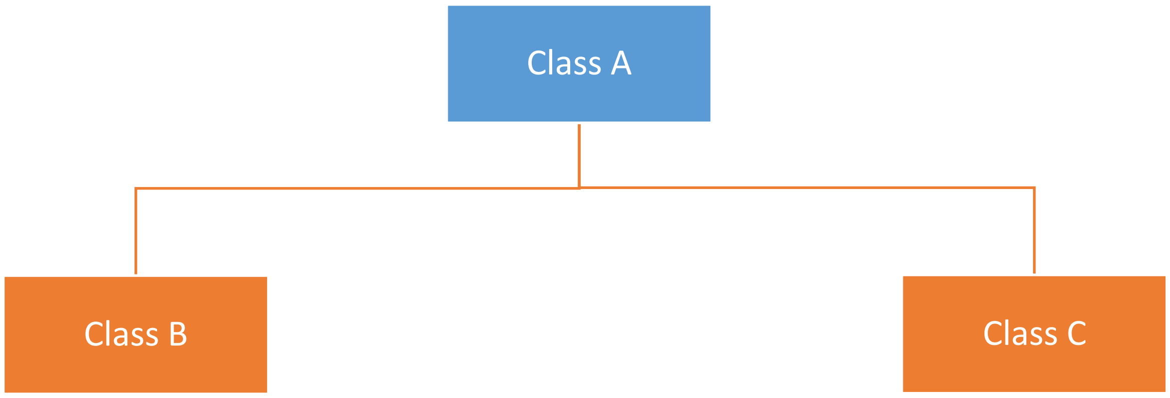
1. **Single Inheritance**



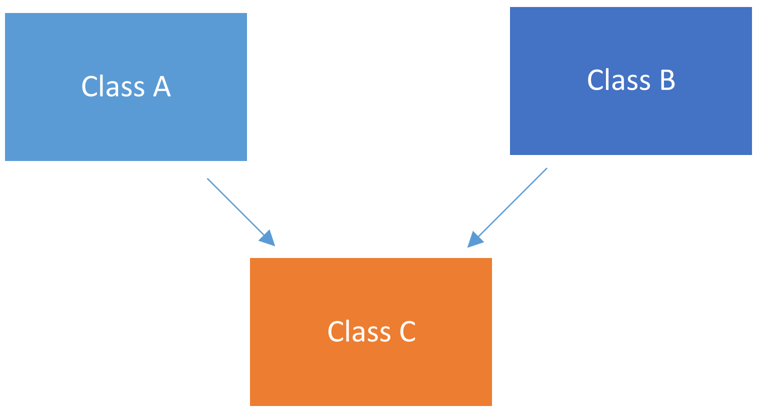
2. **Multi-Level Inheritance**



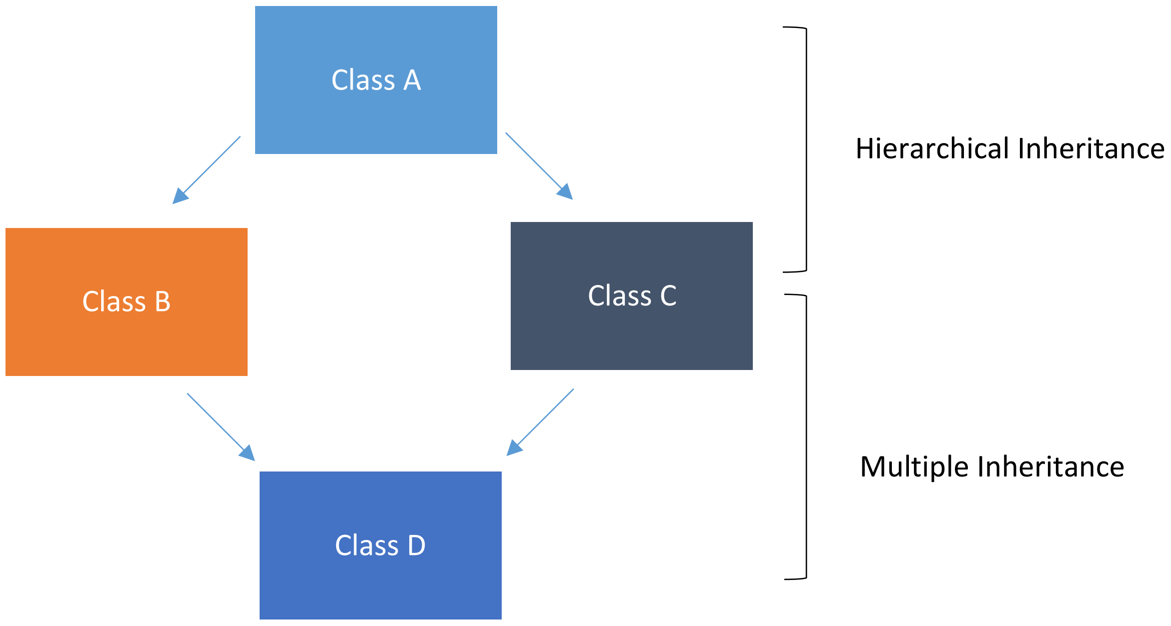
3. **Hierarchical Inheritance**



4. **Multiple Inheritance**



5. **Hybrid Inheritance**
   It is the combination of any two types of inheritance.



Note:

- Constructor of Super class will not inherited to Sub class.
- Private members will not be inherited to Sub class.
- If we don't inherit any class then compiler will inherit it from Object class by default.
  Example: Public class **A** extends **object**
- Every class must extends another class names Object.
- Object class is the Parent class of every class in Java.
- Every class can have maximum one super class, not more than one super class.

Relationship Between Classes

- **Types of Relation between Classes:**
  1. Inheritance (IS-A)
  2. Association (Has-A)
     a. Aggregation
     b. Composition

- **Advantages of Relation between Classes:**
  1. Code Reusability
  2. Cost Cutting
  3. Reduce Redundancy

1. Inheritance (Is-A)

```
Class Animal
{
        Public void eat()
        {
                System.out.println("Animal is Eating");
        }
}

Class Dog extends Animal
{
        Public static void main (String[] args)
        {
                Dog d = new Dog();
                d.eat();
        }
}
```

Dog **<u>Is-A</u>** Animal.

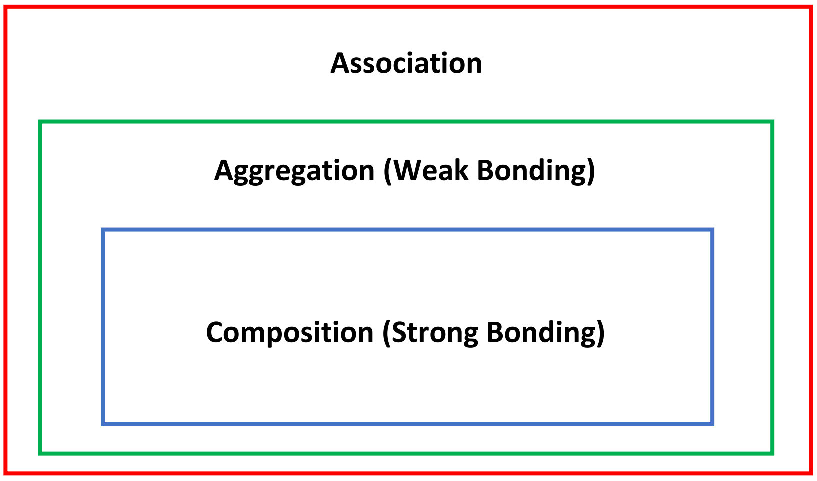
2. Association (Has-A)

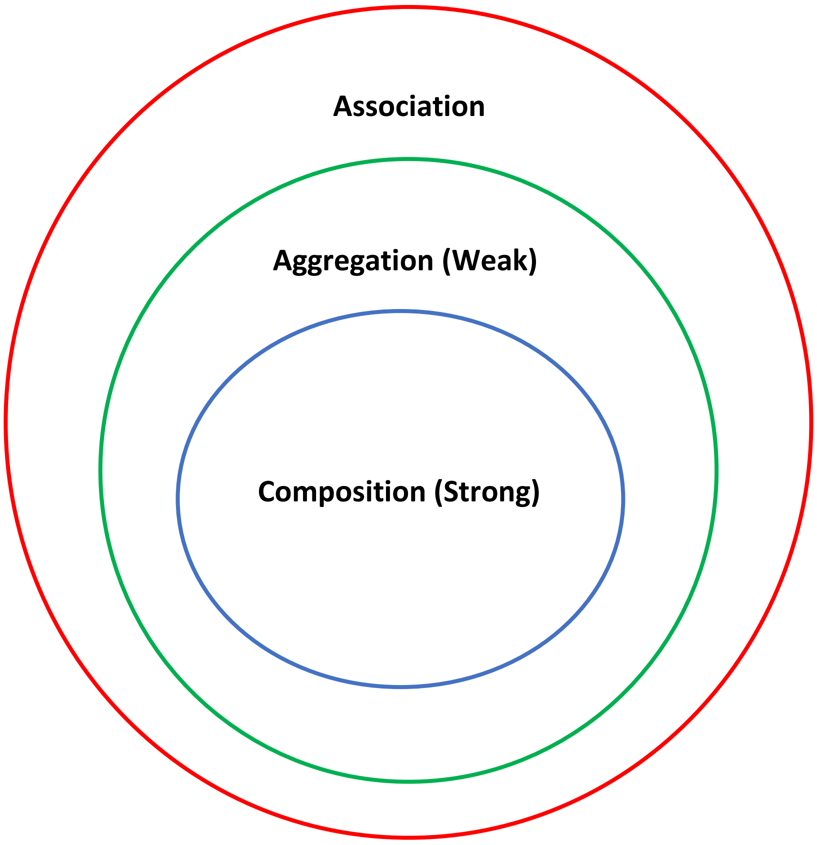
```
Class Student
{
        Int rollNo;
        String name;
}
```

Student **<u>Has-A</u>** rollNo;
Student **<u>Has-A</u>** name;

```
Class Engine
{
}
Class Car
{
        Engine e = new Engine();
}
```

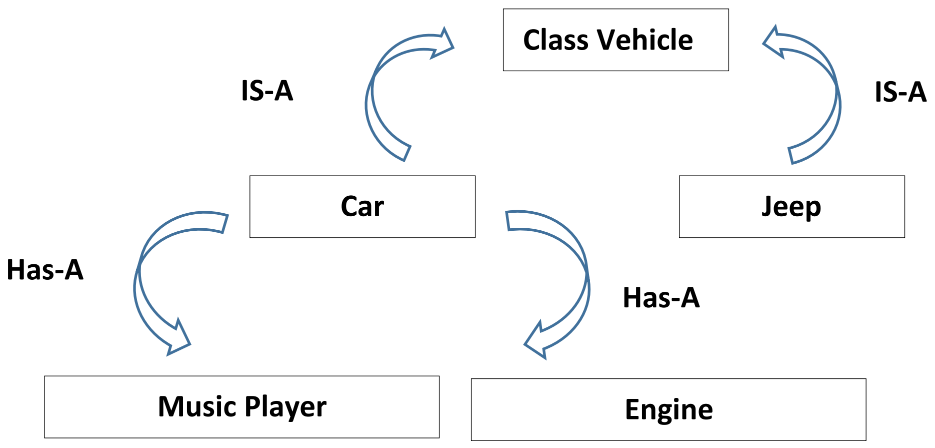
Car **<u>Has-A</u>** Engine;





- Car = Music Player (Weak) – Engine (Strong)
  Car **<u>Has-A</u>** Music Player.
  Car **<u>Has-A</u>** Engine.

Inheritance	Association
Is-A Relationship	Has-A Relationship
Extends Keyword	Reference Variable, New Keyword
Blood Relation	Non-Blood Relation
Classes are Tightly Coupled	Classes are not Tightly Coupled
	Aggression (Weak Bonding)
	Composition (Strong Bonding)
Represent Bottom To Top	Represent Top To Bottom



- If we use extends keyword we get all the properties and methods but when we create its Object we can get selected properties and methods.

Polymorphism

- **Poly** means **Many** and **Morphism** means **Forms**.
- One same thing having different forms or many forms.
- Water: Solid, Liquid, Gas
- Shapes: Circle, Triangles, Rectangles
- Sound: Barking, Roar

Types of Polymorphism

1. Compile Time Polymorphism
- Static Polymorphism
- Method Overloading
- Compiler Handle

2. Run Time Polymorphism
- Dynamic Polymorphism
- Method Overriding
- JVM Handle

1. **Method Overloading**
    1. Same Name of Method
    2. Same Class
    3. Different Arguments
        - Number of Arguments
        - Sequence of Arguments
        - Type of Arguments

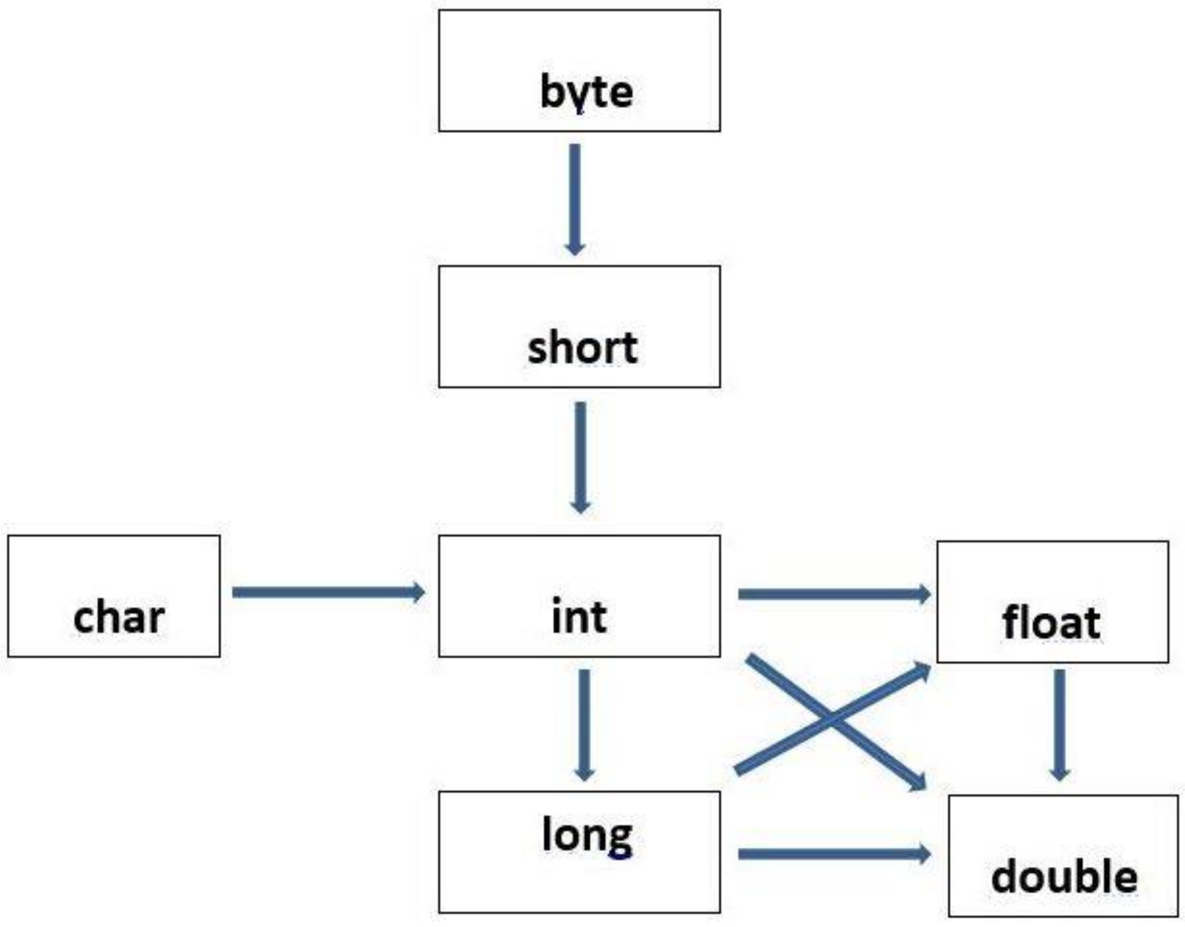
```
Public class Test
{
        Public void show()
        {
                System.out.println("Simple Show Method");
        }
        Public void show(int a)
        {
                System.out.println("Int Method");
        }
        Public void show(int a, float b)
        {
                System.out.println("Int and Float Method");
        }
        Public static void main(String[] args)
        {
                Test t = new Test();
                t.show();
                t.show(10);
                t.show(10, 5.5);
        }
}
```

Interview Questions:

1. **Can we achieve method overloading by changing the return type of method only?**
   In Java, method overloading is not possible by changing the return type of the method only because of ambiguity.

2. **Can we overload Java main() method?**
   Yes, we can have any number of main method in a class by method overloading. This is because JVM always calls main() method which receives String Array as arguments only.

3. **Automatic Type Promotion**
   One type is promoted to another implicitly if no matching data type is found.



2. Method Overriding

1. Same Name of Method
2. Different Class
3. Same Arguments
   - No of Arguments
   - Sequence of Arguments
   - Type of Arguments
4. Inheritance (Is-A)

```
Class Test
{
        Public void show()
        {
                System.out.println("This is Original Method");
        }
}

Class XYZ extends Test
{
        Public void show()
        {
                System.out.println("This is Overridden Method.");
        }

        Public static void main(String[] args)
        {
                Test t = new Test();
                t.show();       // This is Original Method

                XYZ obj = new XYZ();
                Obj.show();     // This is Overridden Method.
        }
}
```

- Use of Method Overriding
  We can change the implementation of parent class in child class.

Interview Questions:

1. **Do overriding method must have same return type or sub type?**
   From Java 5 onwards it is possible to have different return type for a overriding method in child class but child class method return type should be sub type of parent class method return type. This phenomena is known as Covariant return type.

   ```
   Class Test
   {
           Public Object show()
           {
                   System.out.println("This is Object Type Method.");
                   Return null;
           }
   }
   Class XYZ extends Test
   {
           Public String show()
           {
                   System.out.println("This is String Method");
                   Return null;
           }
   }
   ```

2. **Overriding and Access Modifiers.**
   The access modifiers for an overloading method can allow more but no less access than the overridden method. For example, a protected instance method in super class can be made public but not private in the sub class. Doing so will generate the compile time error.

3. **Abstraction and Overriding**
   Abstract methods always and must be override otherwise compile time error will be thrown.

4. **Which Methods Cannot be overridden?**
   Final, Static, Private methods cannot be overridden.

5. **Invoking Overridden Methods From Sub Class.**
   We can call parent class method in overriding method using super keyword.

6. **Overriding and Synchronized/Strictfp Method.**
   The presence of Synchronized/Strictfp modifiers with method have no effect on the rules of overriding. For example it is possible that a Synchronized/Strictfp methods can override a non-Synchronized/Strictfp one and vice-versa.

Abstraction

- Abstraction is hiding internal implementation and just highlighting the main services that we are offering.
- Abstraction is detail hiding or Implementation hiding.
- Example: Car has relevant parts like Steering, Gear, Horn, Accelerator, Break etc are shown to driver but driver does not know the internal functionality of these parts.
- Abstraction can be achieved by two ways
    1. By using Abstract Classes (0% - 100% Abstraction)
    2. By using Interfaces (100% Abstraction)

- **Abstract Class and Methods**
    - A method without body is known as Abstract Method.
    - An abstract method always must be declared in an abstract class.
    - Abstract class can have abstract methods and concrete methods (Methods with body).
    - If a class extends an abstract class then the class must have to implement all the abstract methods of abstract parent class.
    - Abstract methods inside abstract class are meant to be overridden in derived concrete classes otherwise compile time error will be thrown.
    - We cannot create the object of abstract class.

Syntax:

```
abstract class Vehicle {

   int no_of_tyres;

   abstract void start();

   void display() {
      System.out.println("This Is Concrete Method.");
   }
}

class Car extends Vehicle {
   void start() {
      System.out.println("Start With Key.");
   }
}

class Bike extends Vehicle {
   void start() {
      System.out.println("Start With Kick.");
   }
}
```

Interfaces

- Interface is a mechanism to achieve abstraction in java.
- Interfaces are similar to abstract classes but having all the methods of abstract types. For example it cannot have a method with body.
- Interfaces are the blueprint of the class it specify what a class must do but not how to do.
- Since Java 8, we can have default and Static methods in an interface.
- Since Java 9, we can have private methods in an Interface.

Use of Interface

- It is used to achieve abstraction.
- It supports multiple inheritance.
- It can be used to achieve loose coupling.

Syntax:

```
interface InterFaceName {

    // Fields: Only Public Static Final
    public static final int number = 10;

    // Methods: Only Public Abstract.
    public abstract void Method1();

    // If We Don't Write Compiler Will Automatically Add Public Abstract.
    void Method2();

    default void Method3() {
        System.out.println("This Is Method Method 3.");
    }

    public static void Method4() {
        System.out.println("This Is Method Method 4.");
    }

    private void Method5() {
        System.out.println("This Is Method Method 5.");
    }
}
```

Example:

```
interface I1 {

   public abstract void show();
}

class Test implements I1 {

   public void show() {
      System.out.println("This Is Overridden Show Method.");
   }
}

public static void main(String[] args) {

      Test t = new Test();
      t.show();
   }
```

- Multiple Inheritance Using Interfaces

```
interface I1 {

   public abstract void show();
}

interface I2 {

   public abstract void display();
}

class Test implements I1, I2 {

   public void show() {
      System.out.println("This Is Overridden Show Method From I1.");
   }

   public void display() {
      System.out.println("This Is Overridden Display Method From I2.");
   }
}
public static void main(String[] args) {
      Test t = new Test();
      t.show();
      t.display();
   }
```

- **Similarities between Abstract Class and Interfaces**
  Both can contains abstract methods.
  We cannot create an instance or object of abstract class and interface.

Difference between Abstract class and Interfaces

Abstract Class	Interfaces
1. Abstract class can have instance methods that implements a default behavior.	1. Methods of a Java interface are implicitly abstract and cannot have implementations.
2. An abstract class may contain non-final variables.	2. Interfaces contains public, static and final variables only.
3. Methods and Variables can have any access-modifiers i.e. public, protected, default & private.	3. Methods and Variables are always public.
4. Java abstract class should be extended using keyword “extends”.	4. Java interface should be implemented using keyword “implements”.
5. An abstract class can extend another Java class and implement multiple Java interfaces.	5. An interface can extends another Java interface only.

Encapsulation

- ➢ Encapsulation is a mechanism of wrapping the data (Variables) and code acting on the data (Methods) together as a single unit.
- ➢ Steps to achieve Encapsulation:
  1. Declare the variables of class as Private.
  2. Provide Public setter and getter methods to modify and view the variable values.

Example:

```
Class Employee
{
        // Step 01: Data Hiding
        Private int emp_id;

        // Step 02
        Public void setData (int emp_id)
        {
                This.emp_id = emp_id;
        }

        Public int getData ()
        {
                return emp_id;
        }
}
```

- ➢ In Encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class. This concept is known as Data Hiding.
- ➢ Data Hiding is the sub part of encapsulation.

Difference between Encapsulation and Abstraction:

Abstraction	**Encapsulation**
1. Abstraction is detail hiding (Implementation hiding).	1. Encapsulation is data hiding (Information hiding).
2. Data abstraction details with exposing the interface to the user and hiding the details of implementation.	2. Encapsulation groups together data and method that acts upon the data.

This Keyword in Java

- **This** keyword is the reference variable that refers to the current object.

Example:

```
Class Employee
{
    int emp_id;

    Public void setData (int emp_id)
    {
            this.emp_id = emp_id;
    }

    Public int show ()
    {
            System.out.println("Employee ID: " + emp_id);
    }

    Public static void main (String [] args)
    {
            Employee e = new Employee();
            e.setData(10);
            e.show();
    }
}
```