Java OOPs Concepts:

In this article, we will learn about the basics of **OOPs**. Object-Oriented Programming is a **paradigm** that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

The popular object-oriented languages are **Java**, C#, PHP, Python, C++, etc.

The main aim of object-oriented programming is to **implement** real-world **entities**, for example, object, classes, abstraction, inheritance, polymorphism, etc.

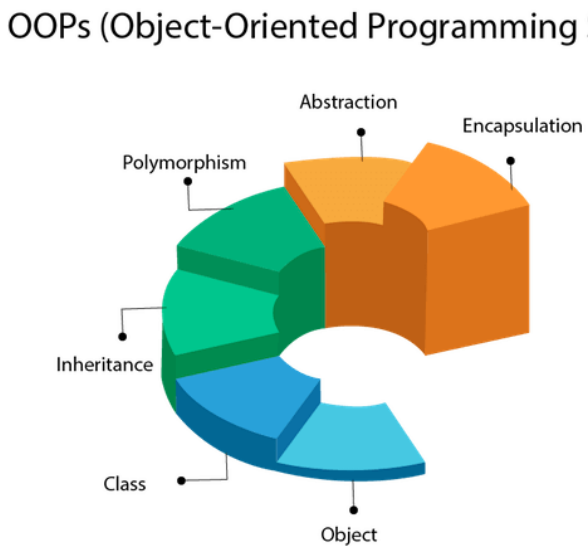
OOPs (Object-Oriented Programming System):

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 by providing some concepts:

- **Inheritance**
- **Polymorphism**
- **Abstraction**
- **Encapsulation**

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- **Coupling**
- **Cohesion**
- **Association**
- **Aggregation**
- **Composition**



Object:

Any entity that has **state** and **behavior** is known as an **object**. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An **Object** can be defined as an **instance** of a class. An object contains an **address** and takes up some **space** in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has **states** like color, name, breed, etc. as well as **behaviors** like wagging the tail, barking, eating, etc.

Class:

*Collection of **objects*** is called class. It is a logical entity.
A class can also be defined as a **blueprint** from which you can create an individual object. Class **doesn't** consume any space.

Inheritance:

*When one object acquires **all** the **properties** and **behaviors** of a parent object*, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism:

If *one task is performed in **different ways***, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
In Java, we use method overloading and method overriding to achieve polymorphism.
Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction:

*Hiding internal details and showing **functionality*** is known as abstraction. For example phone call, we don't know the internal processing.
In Java, we use **abstract** class and **interface** to achieve abstraction.

Encapsulation:

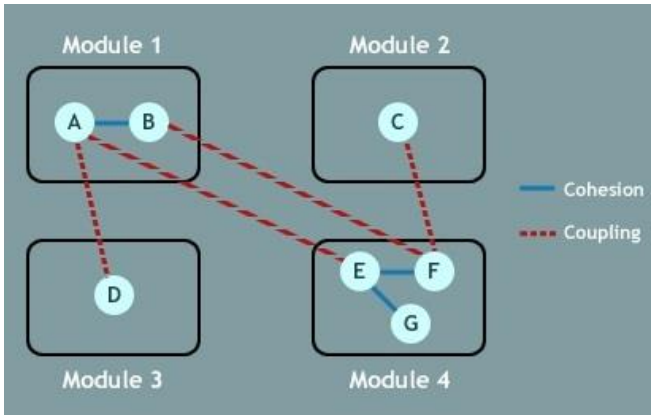
Binding *(*or wrapping*)* code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling:

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion:

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.



Association:

Association **represents** the relationship between the **objects**. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One
- Many to Many

Let's understand the relationship with real-time examples. For example, one country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

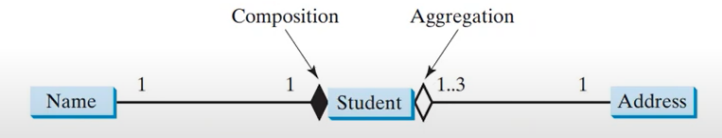
Association can be **unidirectional** or **bidirectional**.

Aggregation:

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the **weak** relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.
Ex. We had two classes **Student** and **Address**. Student has an Address but if we deleted the Student class, Address class will not affected so it is a weak relationship.

Composition:

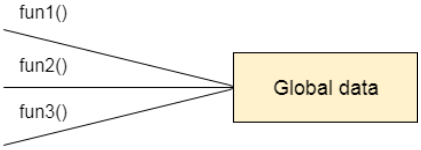
The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a **strong** relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the **parent** object, all the **child** objects will be deleted automatically.



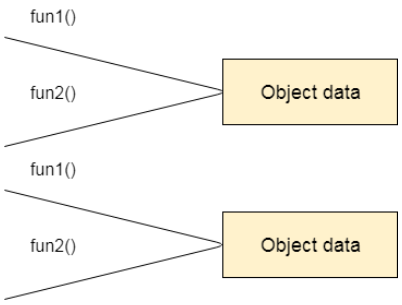
Advantage of OOPs over Procedure-oriented programming language:

1) OOPs makes **development** and maintenance **easier**, whereas, in a **procedure**-oriented programming language, it is **not** easy to manage if code **grows** as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.



Data Representation in Procedure-Oriented Programming



Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

Encapsulation:

Encapsulation in Java is a process of wrapping **code** and **data** together into a **single** unit, for example, a capsule which is mixed of several medicines.

We can create a **fully encapsulated** class in Java by making all the data **members** of the class **private**. Now we can use **setter** and **getter** methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java:

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for **unit** testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Simple Example of **Encapsulation** in Java:

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

File: Student.java

```
1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. package com.javatpoint;
4. public class Student{
5. //private data member
6. private String name;
7. //getter method for name
8. public String getName(){
9. return name;
10. }
11. //setter method for name
12. public void setName(String name){
13. this.name=name
14. }
15. }
```

File: Test.java

```
1. //A Java class to test the encapsulated class.
2. package com.javatpoint;
3. class Test{
4. public static void main(String[] args){
5. //creating instance of the encapsulated class
6. Student s=new Student();
7. //setting value in the name member
8. s.setName("vijay");
9. //getting value of the name member
10. System.out.println(s.getName());
11. }
12. }
```

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output:

Vijay

Read-Only class

```
1. //A Java class which has only getter methods.
2. public class Student{
3. //private data member
4. private String college="AKG";
5. //getter method for college
6. public String getCollege(){
7. return college;  }
8. }
```

Now, you can't change the value of the college data member which is "AKG".

```
s.setCollege("KITE");//will render compile time error
```

Write-Only class

```
1. //A Java class which has only setter methods.
2. public class Student{
3. //private data member
4. private String college;
5. //getter method for college
6. public void setCollege(String college){
7. this.college=college;  }
8. }
```

Now, you can't get the value of the college, you can only change the value of college data member.

Another Example of Encapsulation in Java:

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

File: Account.java

```
1. //A Account class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. class Account {
4. //private data members
5. private long acc_no;
6. private String name,email;
7. private float amount;
8. //public getter and setter methods
9. public long getAcc_no() {
10. return acc_no;
11. }
12. public void setAcc_no(long acc_no) {
13. this.acc_no = acc_no;
14. }
15. public String getName() {
16. return name;
17. }
18. public void setName(String name) {
19. this.name = name;
20. }
21. public String getEmail() {
22. return email;
23. }
24. public void setEmail(String email) {
25. this.email = email;
26. }
27. public float getAmount() {
28. return amount;
29. }
30. public void setAmount(float amount) {
31. this.amount = amount;
32. }


33. }
```

File: TestAccount.java

```
1. //A Java class to test the encapsulated class Account.
2. public class TestEncapsulation {
3. public static void main(String[] args) {
4. //creating instance of Account class
5. Account acc=new Account();
6. //setting values through setter methods
7. acc.setAcc_no(7560504000L);
8. acc.setName("Sonoo Jaiswal");
9. acc.setEmail("sonoojaiswal@javatpoint.com");
10. acc.setAmount(500000f);
11. //getting values through getter methods
12. System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
13. }
14. }
```

Output:

```
7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0
```

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**.

Abstract class in Java:

A **class** which is declared with the **abstract** keyword is known as an abstract class in Java. It can have **abstract** and **non-abstract** methods (method with the body).
Before learning the Java abstract class, let's understand the abstraction in Java first.

Ways to achieve Abstraction:

There are **two** ways to achieve abstraction in java

- Abstract class (0 to 100%) (**Partially** Abstraction)
- Interface (100%) (**Fully** Abstraction)

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**.
- 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.

Example of abstract class

```
abstract class A{}
```

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 and abstract
```

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.

```
1. abstract class Bike{
2. abstract void run();
3. }
4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely");}
6. public static void main(String args[]){
7. Bike obj = new Honda4();
8. obj.run();
9. }
10. }
```

```
running safely
```

Understanding the real scenario of Abstract class:

In this example, 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**.

A **factory method** is a method that returns the **instance** of the class. We will learn about the factory method later.

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

File: TestAbstraction1.java

```
1. abstract class Shape{
2. abstract void draw();  }
3. //In real scenario, implementation is provided by others i.e. unknown by end user
4. class Rectangle extends Shape{
5. void draw(){System.out.println("drawing rectangle");}
6. }
7. class Circle1 extends Shape{
8. void draw(){System.out.println("drawing circle");}
9. }
10. //In real scenario, method is called by programmer or user
11. class TestAbstraction1{
12. public static void main(String args[]){
13. Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() me
    thod
14. s.draw();  }
15.  }
```

```
drawing circle
```

Another example of Abstract class in java:

File: TestBank.java

```
1. abstract class Bank{
2. abstract int getRateOfInterest();
3. }
4. class CIB extends Bank{
5. int getRateOfInterest(){return 7;}
6. }
7. class QNB extends Bank{
8. int getRateOfInterest(){return 8;}
9. }
10. class TestBank{
11. public static void main(String args[]){
12. Bank b;
13. b=new CIB();
14. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
15. b=new QNB();
16. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
17. }}
```

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class having constructor, data member and methods:

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

File: TestAbstraction2.java

```
1. //Example of an abstract class that has abstract and non-abstract methods
2. abstract class Bike{
3. Bike(){System.out.println("bike is created");}
4. abstract void run();
5. void changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a Child class which inherits Abstract class
8. class Honda extends Bike{
9. void run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. class TestAbstraction2{
13. public static void main(String args[]){
14. Bike obj = new Honda();
```

```
15. obj.run();
16. obj.changeGear();  }
17. }
```

```
bike is created
running safely..
gear changed
```

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

```
class Bike12{
abstract void run();
}
```

```
compile time error
```

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

Another real scenario of abstract class:

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

```
1.  interface A{
2.  void a();
3.  void b();
4.  void c();
5.  void d();
6.  }
7.  abstract class B implements A{
8.  public void c(){System.out.println("I am c");}
9.  }
10. class M extends B{
11. public void a(){System.out.println("I am a");}
12. public void b(){System.out.println("I am b");}
13. public void d(){System.out.println("I am d");}
14. }
15. class Test5{
16. public static void main(String args[]){
17. A a=new M();
18. a.a();
19. a.b();
20. a.c();
21. a.d();
22. }}
```

```
Output:  I am a
         I am b
         I am c
         I am d
```

Inheritance:

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).

The idea behind inheritance in Java is that you can **create** new classes that are built upon **existing** classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

Why use inheritance in java?

For Method **Overriding** (so **runtime polymorphism** can be achieved).
For Code Reusability.

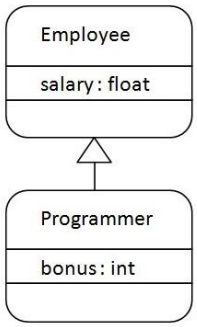
Terms used in Inheritance:

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
Sub Class/Child Class: Subclass is a class which **inherits** the other class. It is also called a **derived** class, **extended** class, or **child** class.
Super Class/Parent Class: Superclass is the class from where a **subclass inherits** the features. It is also called a **base** class or a **parent** class.
Reusability: As the name specifies, reusability is a mechanism which facilitates you to **reuse** the fields and **methods** of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance:

```
class Subclass-name extends Superclass-name
{
//methods and fields}
```

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.



Java Inheritance Example:

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2. float salary=40000;
3. }
4. class Programmer extends Employee{
```

```
5. int bonus=10000;
6. public static void main(String args[]){
7. Programmer p=new Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

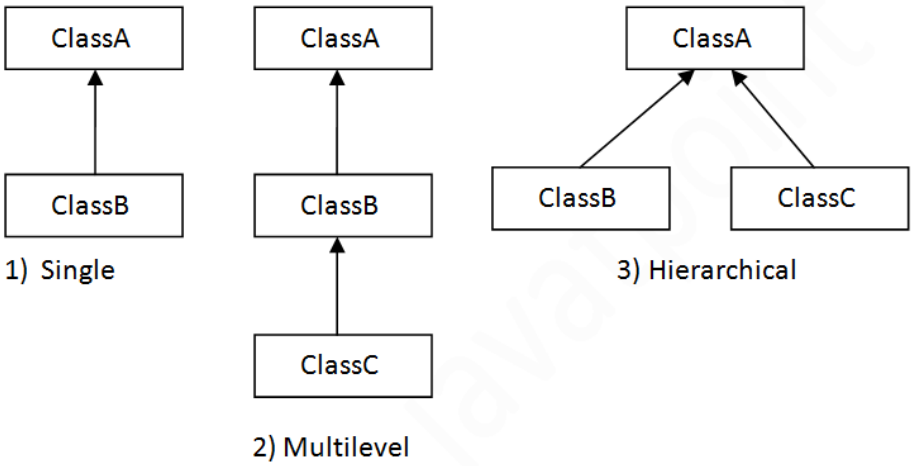
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java:

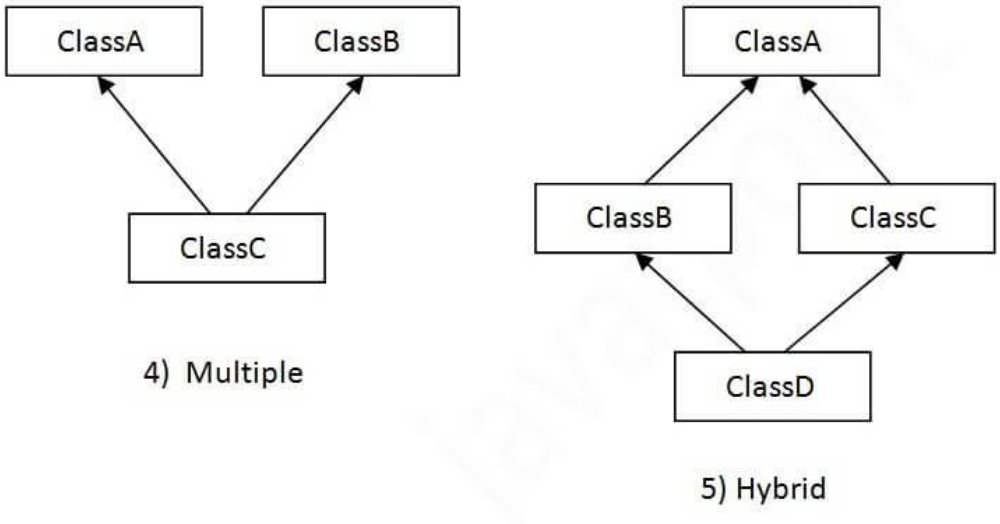
On the basis of class, there can be three types of inheritance in java: **single**, **multilevel** and **hierarchical**.

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



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

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.

File: TestInheritance.java

```
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 Example:

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.

File: TestInheritance2.java

```
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...
```

Hierarchical Inheritance Example:

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.

File: TestInheritance3.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

Output:

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

Q) 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.

```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A,B{//suppose if it were  }
8. public static void main(String args[]){
9. C obj=new C();
10. obj.msg();//Now which msg() method would be invoked?
11. } }
```

```
Compile Time Error
```

Polymorphism:

Polymorphism in Java is a concept by which we can perform a ***single action in different ways***. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means **many** and "morphs" means **forms**. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method **overloading** and method **overriding**.

If you **overload** a **static** method in Java, it is the example of **compile time polymorphism**. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java:

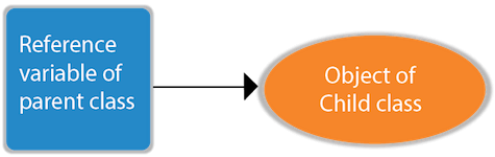
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an **overridden method** is resolved at runtime rather than compile-time.

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.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting:

If the **reference** variable of **Parent** class **refers** to the object of **Child** class, it is known as upcasting. For example:




```
class A{}
class B extends A{}
A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}
class A{}
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism:

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

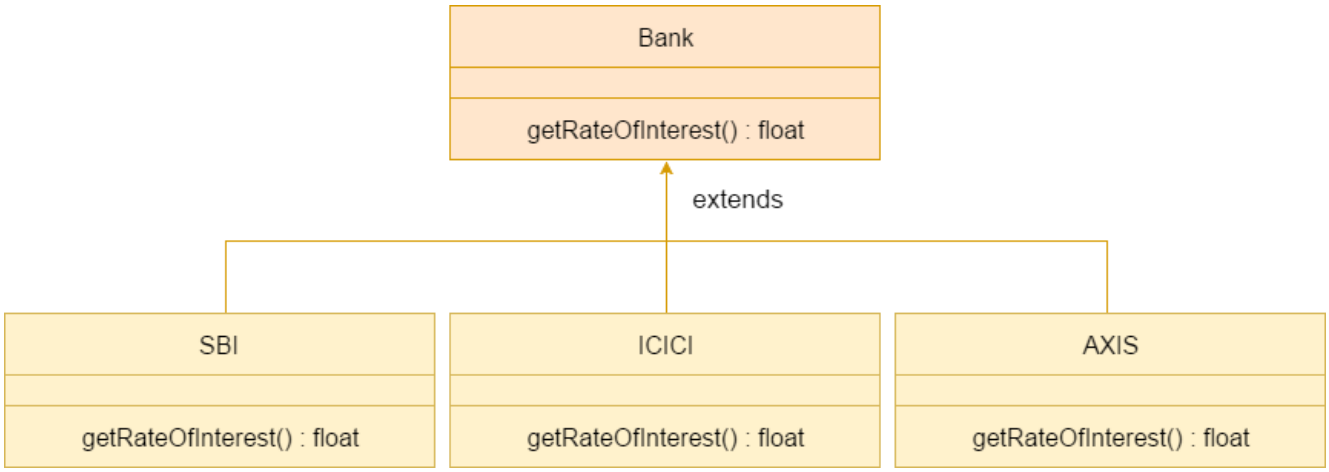
```
1. class Bike{
2. void run(){System.out.println("running");}
3. }
4. class Splendor extends Bike{
5. void run(){System.out.println("running safely with 60km");}

6. public static void main(String args[]){
7. Bike b = new Splendor();//upcasting
8. b.run();
9. }
10. }
```

Output:

```
running safely with 60km.
```

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

```
1. class Bank{
2. float getRateOfInterest(){return 0;}
3. }
4. class SBI extends Bank{
5. float getRateOfInterest(){return 8.4f;}
6. }
7. class ICICI extends Bank{
8. float getRateOfInterest(){return 7.3f;}
9. }
10. class AXIS extends Bank{
11. float getRateOfInterest(){return 9.7f;}
12. }
13. class TestPolymorphism{
14. public static void main(String args[]){
15. Bank b;
16. b=new SBI();
17. System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
18. b=new ICICI();
19. System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
20. b=new AXIS();
21. System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
22. }
23. }
```

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Java Runtime Polymorphism Example: Shape

```
1. class Shape{
2. void draw(){System.out.println("drawing...");}
3. }
4. class Rectangle extends Shape{
5. void draw(){System.out.println("drawing rectangle...");}
6. }
7. class Circle extends Shape{
8. void draw(){System.out.println("drawing circle...");}
9. }
```

```
10. class Triangle extends Shape{
11. void draw(){System.out.println("drawing triangle...");}
12. }
13. class TestPolymorphism2{
14. public static void main(String args[]){
15. Shape s;
16. s=new Rectangle();
17. s.draw();
18. s=new Circle();
19. s.draw();
20. s=new Triangle();
21. s.draw();
22. } }
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Java Runtime Polymorphism Example: Animal

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. }
7. class Cat extends Animal{
8. void eat(){System.out.println("eating rat...");}
9. }
10. class Lion extends Animal{
11. void eat(){System.out.println("eating meat...");}
12. }
13. class TestPolymorphism3{
14. public static void main(String[] args){
15. Animal a;
16. a=new Dog();
17. a.eat();
18. a=new Cat();
19. a.eat();
20. a=new Lion();
```

```
21. a.eat();
22. }}
```

Output:

```
eating bread...
eating rat...
eating meat...
```

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```
1. class Bike{
2. int speedlimit=90;
3. }
4. class Honda3 extends Bike{
5. int speedlimit=150;


6. public static void main(String args[]){
7. Bike obj=new Honda3();
8. System.out.println(obj.speedlimit);//90
9. }
```

Output:

```
90
```

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of **Runtime Polymorphism** with **multilevel** inheritance.

```
1. class Animal{
2. void eat(){System.out.println("eating");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating fruits");}
6. }
7. class BabyDog extends Dog{
8. void eat(){System.out.println("drinking milk");}
```

```
9.  public static void main(String args[]){
10. Animal a1,a2,a3;
11. a1=new Animal();
12. a2=new Dog();
13. a3=new BabyDog();
14. a1.eat();
15. a2.eat();
16. a3.eat();
17. }
18. }
```

Output:

```
eating
eating fruits
drinking Milk
```

Try for Output

```
1.  class Animal{
2.  void eat(){System.out.println("animal is eating...");}
3.  }
4.  class Dog extends Animal{
5.  void eat(){System.out.println("dog is eating...");}
6.  }
7.  class BabyDog1 extends Dog{
8.  public static void main(String args[]){
9.  Animal a=new BabyDog1();
10. a.eat();
11. }}
```

Output:

Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of **Dog** class is invoked.

References:

Java how to program tenth edition: paul deitel, Harvey deitel
javaTpoint
tutorialsPoint