(39 questions)

Basic OOPs Interview Questions

1. What is meant by the term OOPs?

It is the programming paradigm that is defined using objects. Objects can be considered as real-world instances of entities like class, that have some characteristics and behaviors.

2. What are the main features of OOPs?

- Inheritance
- Encapsulation
- Polymorphism
- Data Abstraction

3. What are some advantages of using OOPs?

- OOPs is very helpful in solving very complex level of problems.
- Highly complex programs can be created, handled, and maintained easily using object-oriented programming.
- OOPs, promote code reuse, thereby reducing redundancy.
- OOPs also helps to hide the unnecessary details with the help of Data Abstraction.
- OOPs, are based on a bottom-up approach, unlike the Structural programming paradigm, which uses a top-down approach.
- Polymorphism offers a lot of flexibility in OOPs.

Advanced OOPs Interview Questions

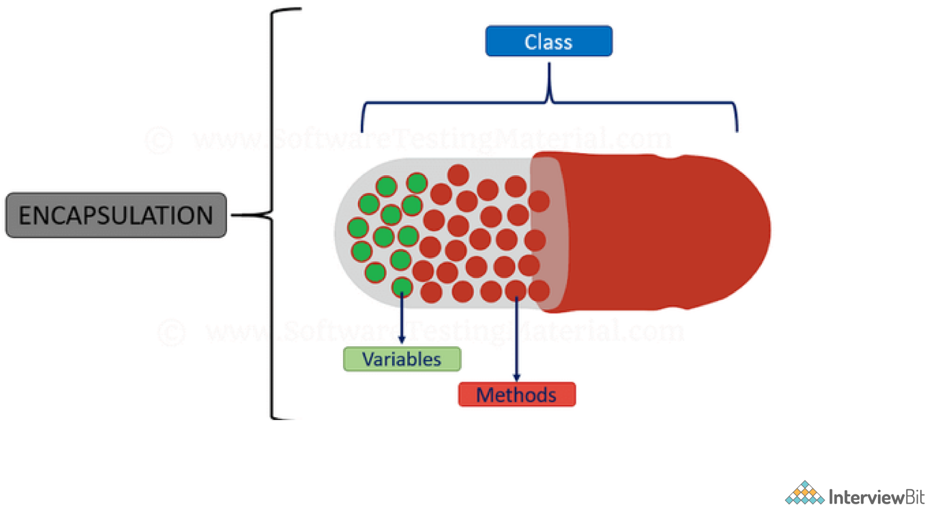
4. What is a class?

A class can be understood as a template or a blueprint, which contains some values, known as member data or member, and some set of rules, known as behaviors or functions. So when an object is created, it automatically takes the data and functions that are defined in the class.Therefore the class is basically a template or blueprint for objects. Also one can create as many objects as they want based on a class.For example, first, a car's template is created. Then multiple units of car are created based on that template.

5. What is an object?

An object refers to the instance of the class, which contains the instance of the members and behaviors defined in the class template. In the real world, an object is an actual entity to which a user interacts, whereas class is just the blueprint for that object. So the objects consume space and have some characteristic behavior.For example, a specific car.

6. What is encapsulation?



One can visualize Encapsulation as the method of putting everything that is required to do the job, inside a capsule and presenting that capsule to the user. What it means is that by Encapsulation, all the necessary data and methods are bound together and all the unnecessary details are hidden from the normal user. So Encapsulation is the process of binding data members and methods of a program together to do a specific job, without revealing unnecessary details.

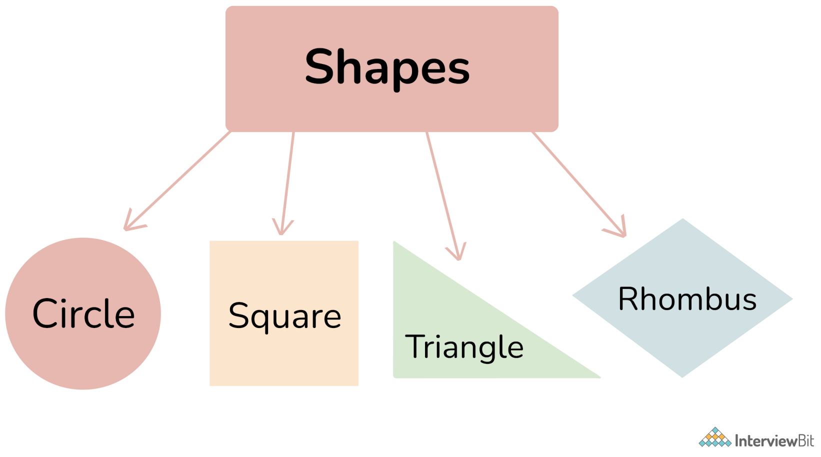
Encapsulation can also be defined in two different ways:

1) Data hiding: Encapsulation is the process of hiding unwanted information, such as restricting access to any member of an object.

2) Data binding: Encapsulation is the process of binding the data members and the methods together as a whole, as a class.

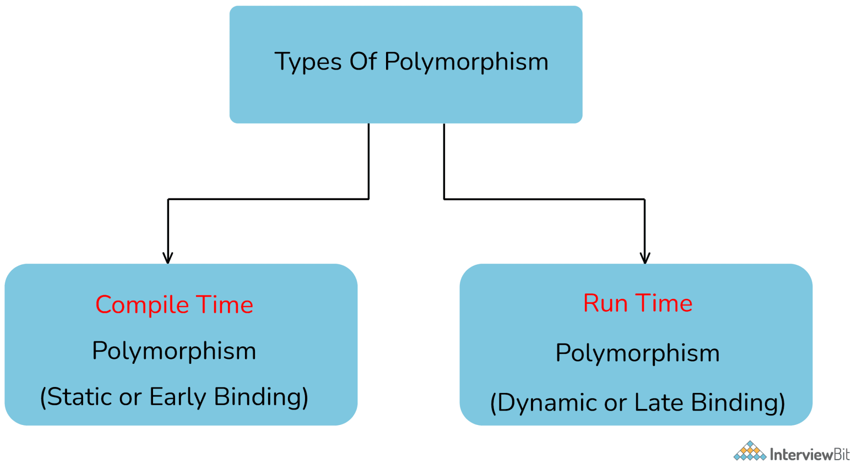
7. What is Polymorphism?

Polymorphism is composed of two words - “poly” which means “many”, and “morph” which means “shapes”. Therefore Polymorphism refers to something that has many shapes.



In OOPs, Polymorphism refers to the process by which some code, data, method, or object behaves differently under different circumstances or contexts. Compile-time polymorphism and Run time polymorphism are the two types of polymorphisms in OOPs languages.

8. What is Compile time Polymorphism and how is it different from Runtime Polymorphism?


Types Of Polymorphism
Compile Time Polymorphism (Static or Early Binding)
Run Time Polymorphism (Dynamic or Late Binding)
InterviewBit

Compile Time Polymorphism: Compile time polymorphism, also known as Static Polymorphism, refers to the type of Polymorphism that happens at compile time. What it means is that the compiler decides what shape or value has to be taken by the entity in the picture.

Example:

```
// In this program, we will see how multiple functions are created with the same name,
// but the compiler decides which function to call easily at the compile time itself.
class CompileTimePolymorphism{
   // 1st method with name add
   public int add(int x, int y){
   return x+y;
```

```
    }
    // 2nd method with name add
    public int add(int x, int y, int z){
    return x+y+z;
    }
    // 3rd method with name add
    public int add(double x, int y){
    return (int)x+y;
    }
    // 4th method with name add
    public int add(int x, double y){
    return x+(int)y;
    }
}
class Test{
    public static void main(String[] args){
    CompileTimePolymorphism demo=new CompileTimePolymorphism();
    // In the below statement, the Compiler looks at the argument
types and decides to call method 1
    System.out.println(demo.add(2,3));
    // Similarly, in the below statement, the compiler calls
method 2
    System.out.println(demo.add(2,3,4));
    // Similarly, in the below statement, the compiler calls
method 4
    System.out.println(demo.add(2,3.4));
    // Similarly, in the below statement, the compiler calls
method 3
    System.out.println(demo.add(2.5,3));
    }
}
```

In the above example, there are four versions of add methods. The first method takes two parameters while the second one takes three. For the third and fourth methods, there is a change of order of parameters. The compiler looks at the method signature and decides which method to invoke for a particular method call at compile time.

Runtime Polymorphism: Runtime polymorphism, also known as Dynamic Polymorphism, refers to the type of Polymorphism that happens at the run time. What it means is it can't be decided by the compiler. Therefore what shape or value has to be taken depends upon the execution. Hence the name Runtime Polymorphism.

Example:

```
class AnyVehicle{
   public void move(){
   System.out.println("Any vehicle should move!!");
   }
}
class Bike extends AnyVehicle{
   public void move(){
   System.out.println("Bike can move too!!");
   }
}
class Test{
   public static void main(String[] args){
   AnyVehicle vehicle = new Bike();
   // In the above statement, as you can see, the object vehicle
is of type AnyVehicle
   // But the output of the below statement will be "Bike can
move too!!",
   // because the actual implementation of object 'vehicle' is
decided during runtime vehicle.move();
   vehicle = new AnyVehicle();
   // Now, the output of the below statement will be "Any
vehicle should move!!",
   vehicle.move();
   }
}
```

As the method to call is determined at runtime, as shown in the above code, this is called runtime polymorphism.

9. What is meant by Inheritance?

The term “inheritance” means “receiving some quality or behavior from a parent to an offspring.” In object-oriented programming, inheritance is the mechanism by which an object or class (referred to as a child) is created using the definition of another object or class (referred to as a parent). Inheritance not only helps to keep the implementation simpler but also helps to facilitate code reuse.

10. What is Abstraction?

If you are a user, and you have a problem statement, you don't want to know how the components of the software work, or how it's made. You only want to know how the software solves your problem. Abstraction is the method of hiding unnecessary details from the necessary ones. It is one of the main features of OOPs.

For example, consider a car. You only need to know how to run a car, and not how the wires are connected inside it. This is obtained using Abstraction.

11. How much memory does a class occupy?

Classes do not consume any memory. They are just a blueprint based on which objects are created. Now when objects are created, they actually initialize the class members and methods and therefore consume memory.

12. Is it always necessary to create objects from class?

No. An object is necessary to be created if the base class has non-static methods. But if the class has static methods, then objects don’t need to be created. You can call the class method directly in this case, using the class name.

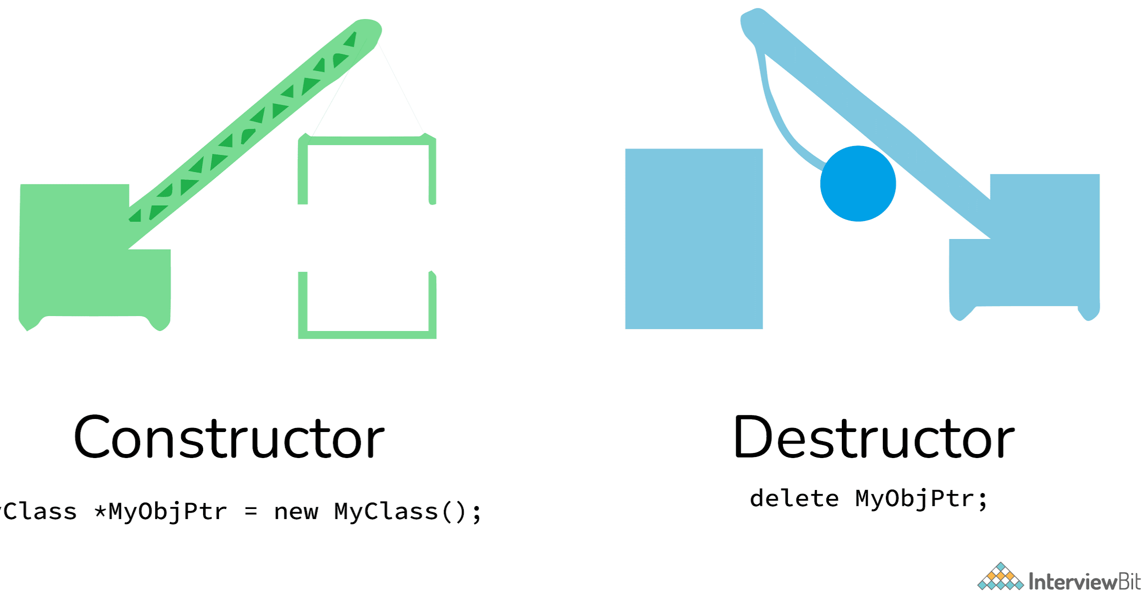
13. What is a constructor?

Constructors are special methods whose name is the same as the class name. The constructors serve the special purpose of initializing the objects.

For example, suppose there is a class with the name "MyClass", then when you instantiate this class, you pass the syntax:

```
MyClass myClassObject = new MyClass();
```

Now here, the method called after "new" keyword - MyClass(), is the constructor of this class. This will help to instantiate the member data and methods and assign them to the object myClassObject.



14. What are the various types of constructors in C++?

The most common classification of constructors includes:

Default constructor: The default constructor is the constructor which doesn't take any argument. It has no parameters.

```
class ABC
```

```
{
   int x;

   ABC()
   {
       x = 0;
   }
}
```

Parameterized constructor: The constructors that take some arguments are known as parameterized constructors.

```
class ABC
{
   int x;

   ABC(int y)
   {
       x = y;
   }
}
```

Copy constructor: A copy constructor is a member function that initializes an object using another object of the same class.

```
class ABC
{
   int x;

   ABC(int y)
   {
       x = y;
   }
   // Copy constructor
   ABC(ABC abc)
   {
       x = abc.x;
   }
}
```

15. What is a copy constructor?

Copy Constructor is a type of constructor, whose purpose is to copy an object to another. What it means is that a copy constructor will clone an object and its values, into another object, is provided that both the objects are of the same class.

16. What is a destructor?

Contrary to constructors, which initialize objects and specify space for them, Destructors are also special methods. But destructors free up the resources and memory occupied by an object. Destructors are automatically called when an object is being destroyed.

17. Are class and structure the same? If not, what's the difference between a class and a structure?

No, class and structure are not the same. Though they appear to be similar, they have differences that make them apart. For example, the structure is saved in the stack memory, whereas the class is saved in the heap memory. Also, Data Abstraction cannot be achieved with the help of structure, but with class, Abstraction is majorly used.

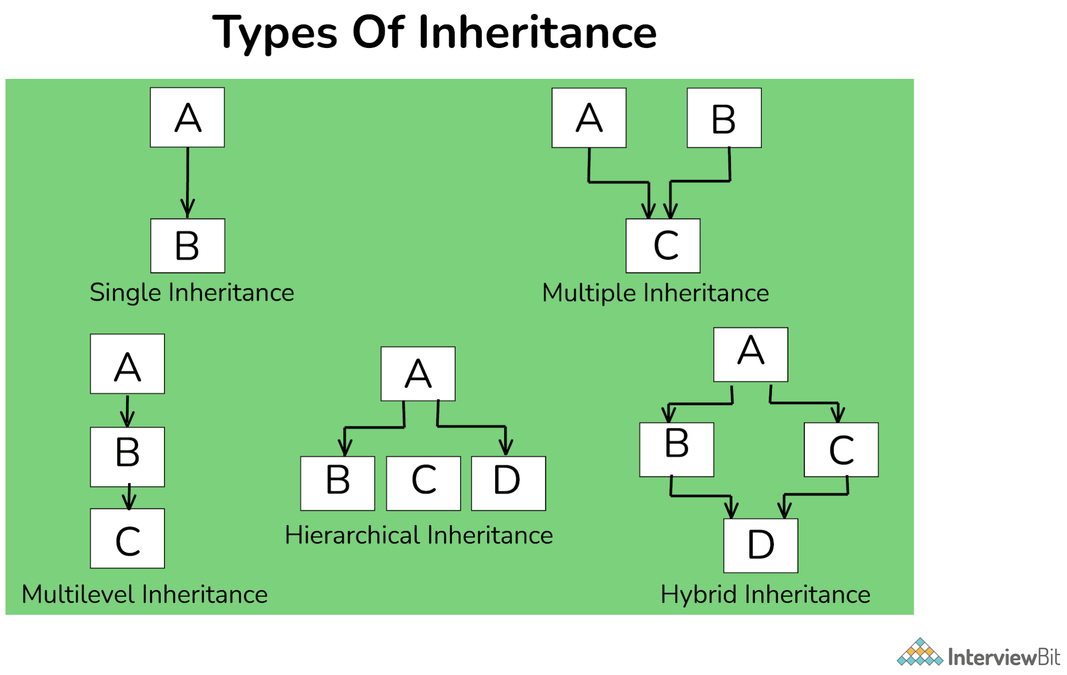
18. Are there any limitations of Inheritance?

Yes, with more powers comes more complications. Inheritance is a very powerful feature in OOPs, but it has some limitations too. Inheritance needs more time to process, as it needs to navigate through multiple classes for its implementation. Also, the classes involved in Inheritance - the base class and the child class, are very tightly coupled together. So if one needs to make some changes, they might need to do nested changes in both classes. Inheritance might be complex for implementation, as well. So if not correctly implemented, this might lead to unexpected errors or incorrect outputs.

19. What are the various types of inheritance?

The various types of inheritance include:

- Single inheritance
- Multiple inheritances
- Multi-level inheritance
- Hierarchical inheritance
- Hybrid inheritance

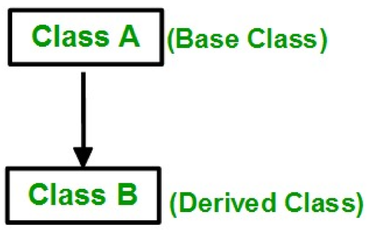


20. What is a subclass?

The subclass is a part of Inheritance. The subclass is an entity, which inherits from another class. It is also known as the child class.

21. Define a superclass?

Superclass is also a part of Inheritance. The superclass is an entity, which allows subclasses or child classes to inherit from itself.



22. What is an interface?

An interface refers to a special type of class, which contains methods, but not their definition. Only the declaration of methods is allowed inside an interface. To use an interface, you cannot create objects. Instead, you need to implement that interface and define the methods for their implementation.

23. What is meant by static polymorphism?

Static Polymorphism is commonly known as the Compile time polymorphism. Static polymorphism is the feature by which an object is linked with the respective function or operator based on the values during the compile time. Static or Compile time Polymorphism can be achieved through Method overloading or operator overloading.

24. What is meant by dynamic polymorphism?

Dynamic Polymorphism or Runtime polymorphism refers to the type of Polymorphism in OOPs, by which the actual implementation of the function is decided during the runtime or execution. The dynamic or runtime polymorphism can be achieved with the help of method overriding.

25. What is the difference between overloading and overriding?

Overloading is a compile-time polymorphism feature in which an entity has multiple implementations with the same name. For example, Method overloading and Operator overloading.

Whereas Overriding is a runtime polymorphism feature in which an entity has the same name, but its implementation changes during execution. For example, Method overriding.

26. How is data abstraction accomplished?

Data abstraction is accomplished with the help of abstract methods or abstract classes.

27. What is an abstract class?

An abstract class is a special class containing abstract methods. The significance of abstract class is that the abstract methods inside it are not implemented and only declared. So as a result, when a subclass inherits the abstract class and needs to use its abstract methods, they need to define and implement them.

28. How is an abstract class different from an interface?

Interface and abstract class both are special types of classes that contain only the methods declaration and not their implementation. But the interface is entirely different from an abstract class. The main difference between the two is that, when an interface is implemented, the subclass must define all its methods and provide its implementation. Whereas when an abstract class is inherited, the subclass does not need to provide the definition of its abstract method, until and unless the subclass is using it.

Also, an abstract class can contain abstract methods as well as non-abstract methods.

29. What are access specifiers and what is their significance?

Access specifiers, as the name suggests, are a special type of keywords, which are used to control or specify the accessibility of entities like classes, methods, etc. Some of the access specifiers or access modifiers include “private”, “public”, etc. These access specifiers also play a very vital role in achieving Encapsulation - one of the major features of OOPs.

30. What is an exception?

An exception can be considered as a special event, which is raised during the execution of a program at runtime, that brings the execution to a halt. The reason for the exception is mainly due to a position in the program, where the user wants to do something for which the program is not specified, like undesirable input.

31. What is meant by Garbage Collection in OOPs world?

Object-oriented programming revolves around entities like objects. Each object consumes memory and there can be multiple objects of a class. So if these objects and their memories are not handled properly, then it might lead to certain memory-related errors and the system might fail.

Garbage collection refers to this mechanism of handling the memory in the program. Through garbage collection, the unwanted memory is freed up by removing the objects that are no longer needed.

OOPs Coding Problems

32. What is the output of the below code?

```
#include<iostream>

using namespace std;
class BaseClass1 {
public:
    BaseClass1()
    { cout << " BaseClass1 constructor called" << endl;   }
};

class BaseClass2 {
public:
    BaseClass2()
    { cout << "BaseClass2 constructor called" << endl;   }
};

class DerivedClass: public BaseClass1, public BaseClass2 {
  public:
   DerivedClass()
    {  cout << "DerivedClass constructor called" << endl;   }
};

int main()
{
  DerivedClass derived_class;
  return 0;
}
```

Output:

```
BaseClass1 constructor called
BaseClass2 constructor called
DerivedClass constructor called
```

Reason:

The above program demonstrates Multiple inheritances. So when the Derived class’s constructor is called, it automatically calls the Base class's constructors from left to right order of inheritance.

33. What will be the output of the below code?

```
class Scaler
{
   static int i;

   static
   {
       System.out.println(“a”);

       i = 100;
   }
}

public class StaticBlock
{
   static
   {
       System.out.println(“b”);
   }

   public static void main(String[] args)
   {
       System.out.println(“c”);

       System.out.println(Scaler.i);
   }
}
```

Output:

```
b
c
a
100
```

Reason:

Firstly the static block inside the main-method calling class will be implemented. Hence 'b' will be printed first. Then the main method is called, and now the sequence is kept as expected.

34. Predict the output?

```
#include<iostream>
using namespace std;

class ClassA {
public:
   ClassA(int ii = 0) : i(ii) {}
   void show() { cout << "i = " << i << endl;}
private:
   int i;
};

class ClassB {
public:
   ClassB(int xx) : x(xx) {}
   operator ClassA() const { return ClassA(x); }
private:
   int x;
};

void g(ClassA a)
{  a.show(); }

int main() {
 ClassB b(10);
 g(b);
 g(20);
 getchar();
 return 0;
}
```

Output:

```
i = 10
i = 20
```

Reason:

ClassA contains a conversion constructor. Due to this, the objects of ClassA can have integer values. So the statement g(20) works. Also, ClassB has a conversion operator overloaded. So the statement g(b) also works.

35. What will be the output in below code?

```
public class Demo{
   public static void main(String[] arr){
         System.out.println("Main1");
   }
   public static void main(String arr){
         System.out.println("Main2");
   }
}
```

Output:

```
Main1
```

Reason:

Here the main() method is overloaded. But JVM only understands the main method which has a String[] argument in its definition. Hence Main1 is printed and the overloaded main method is ignored.

36. Predict the output?

```
#include<iostream>
using namespace std;

class BaseClass{
   int arr[10];
};

class DerivedBaseClass1: public BaseClass { };

class DerivedBaseClass2: public BaseClass { };

class DerivedClass: public DerivedBaseClass1, public
DerivedBaseClass2{};

int main(void)
{
 cout<<sizeof(DerivedClass);
 return 0;
}
```

Output:

```
If the size of the integer is 4 bytes, then the output will be
80.
```

Reason:

Since DerivedBaseClass1 and DerivedBaseClass1 both inherit from class BaseClass, DerivedClass contains two copies of BaseClass. Hence it results in wastage of space and a large size output. It can be reduced with the help of a virtual base class.

37. What is the output of the below program?

```
#include<iostream>

using namespace std;
class A {
public:
  void print()
  { cout <<" Inside A::"; }
};

class B : public A {
public:
  void print()
  { cout <<" Inside B"; }
};

class C: public B {
};

int main(void)
{
 C c;

 c.print();
 return 0;
}
```

Output:

```
Inside B
```

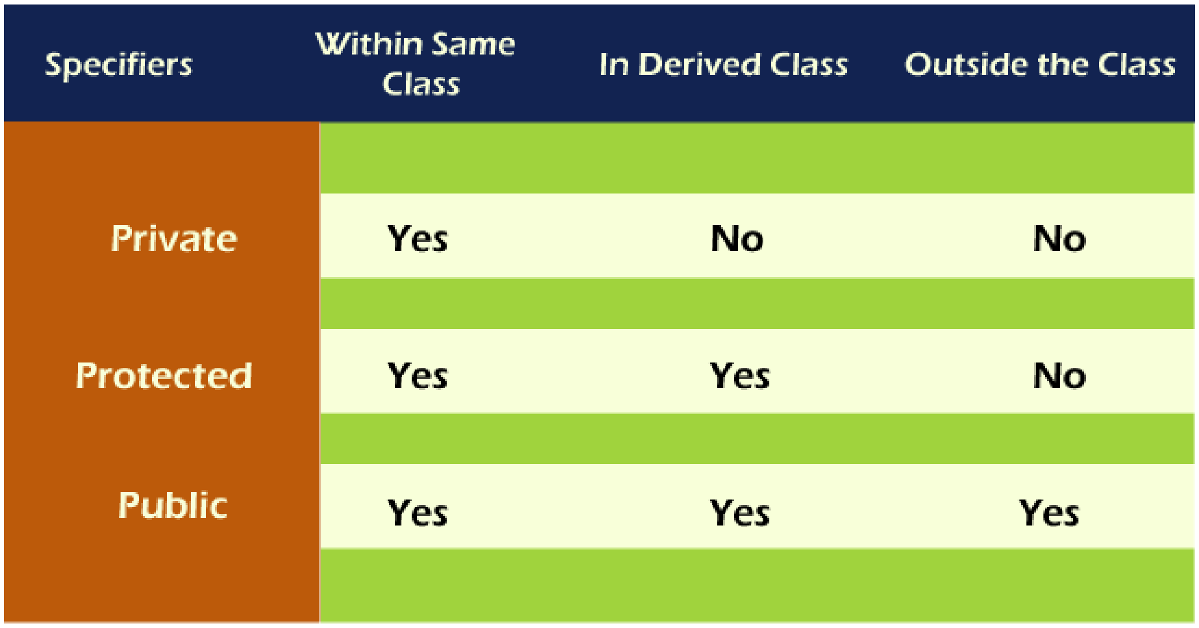
Reason:

The above program implements a Multi-level hierarchy. So the program is linearly searched up until a matching function is found. Here, it is present in both classes A and B. So class B’s print() method is called.

---

38) What is the concept of access specifiers when should we use these?

In OOPs language, **access specifiers** are reserved keyword that is used to set the accessibility of the classes, methods and other members of the class. It is also known as **access modifiers**. It includes **public, private,** and **protected**. There is some other access specifier that is language-specific. Such as Java has another access specifier **default**. These access specifiers play a vital role in achieving one of the major functions of OOP, i.e. encapsulation. The following table depicts the accessibility.



Specifiers	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

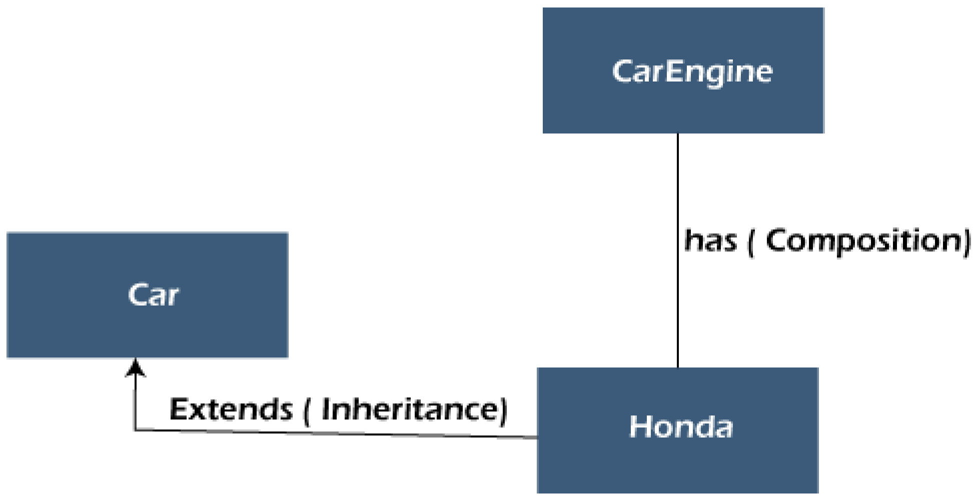
A group of 5 friends, one boy never gives any contribution when the group goes for the outing. Suddenly a beautiful girl joins the same group. The boy who never contributes is now spending a lot of money for the group.

Runtime Polymorphism

---

39) What is composition?

Composition is one of the vital concepts in OOP. It describes a class that references one or more objects of other classes in instance variables. It allows us to model a has-a association between objects. We can find such relationships in the real world. For example, a car has an engine. the following figure depicts the same



The main benefits of composition are:

- Reuse existing code
- Design clean APIs
- Change the implementation of a class used in a composition without adapting any external clients.