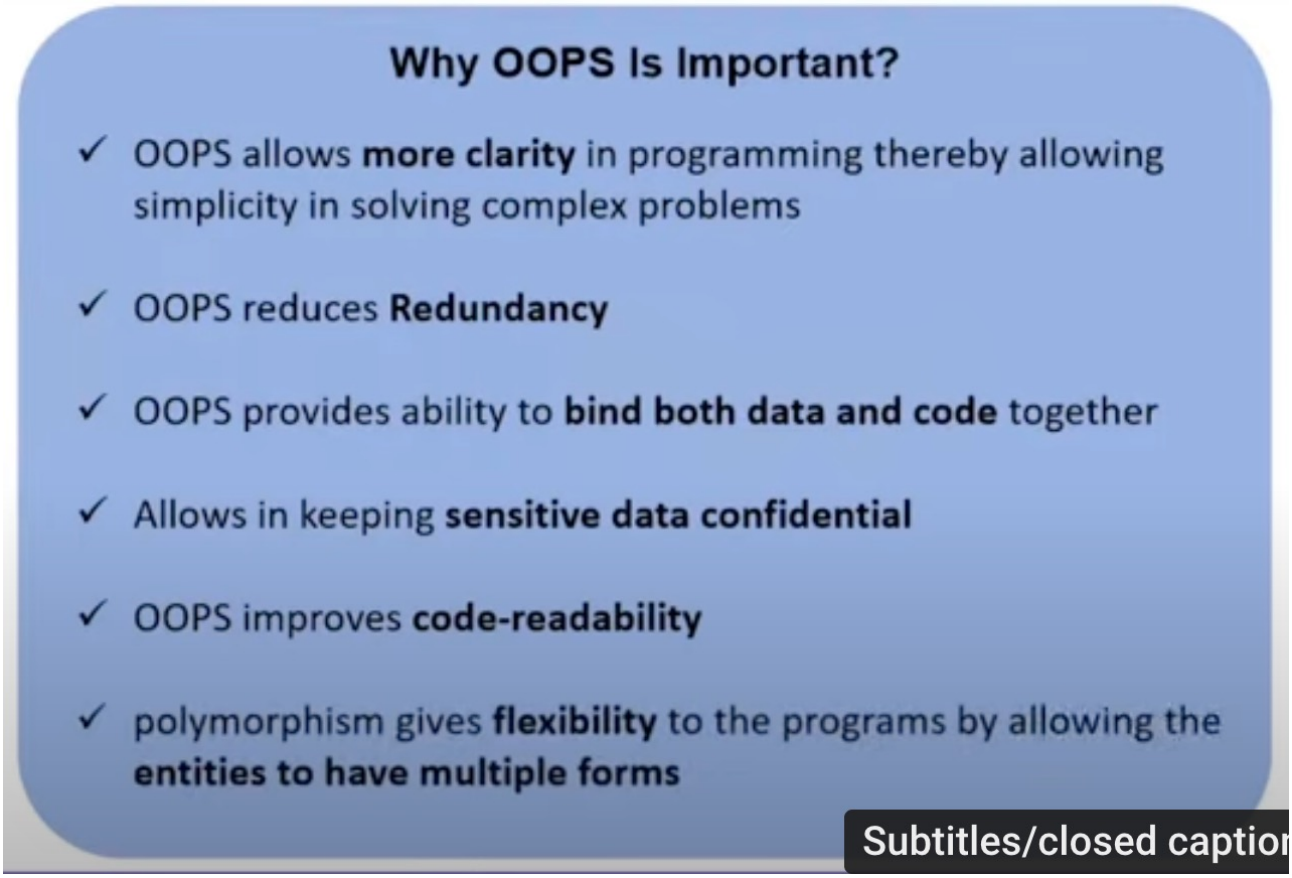
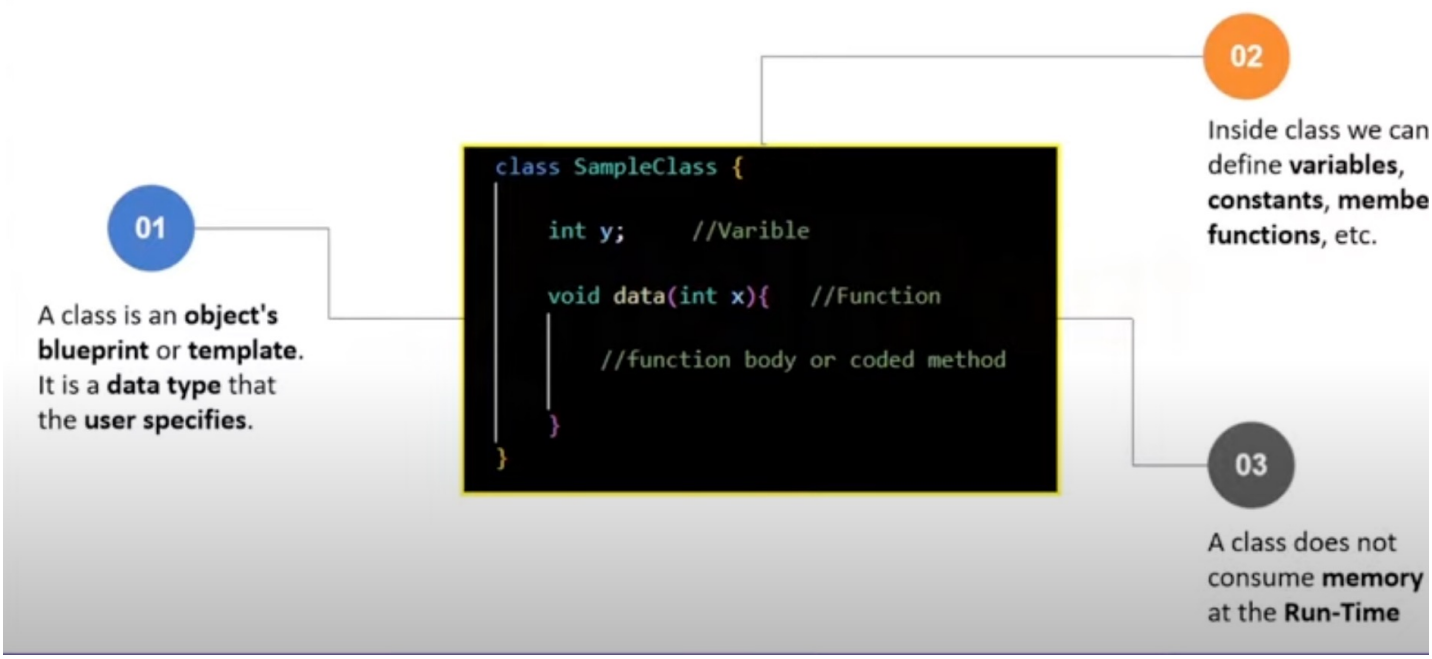
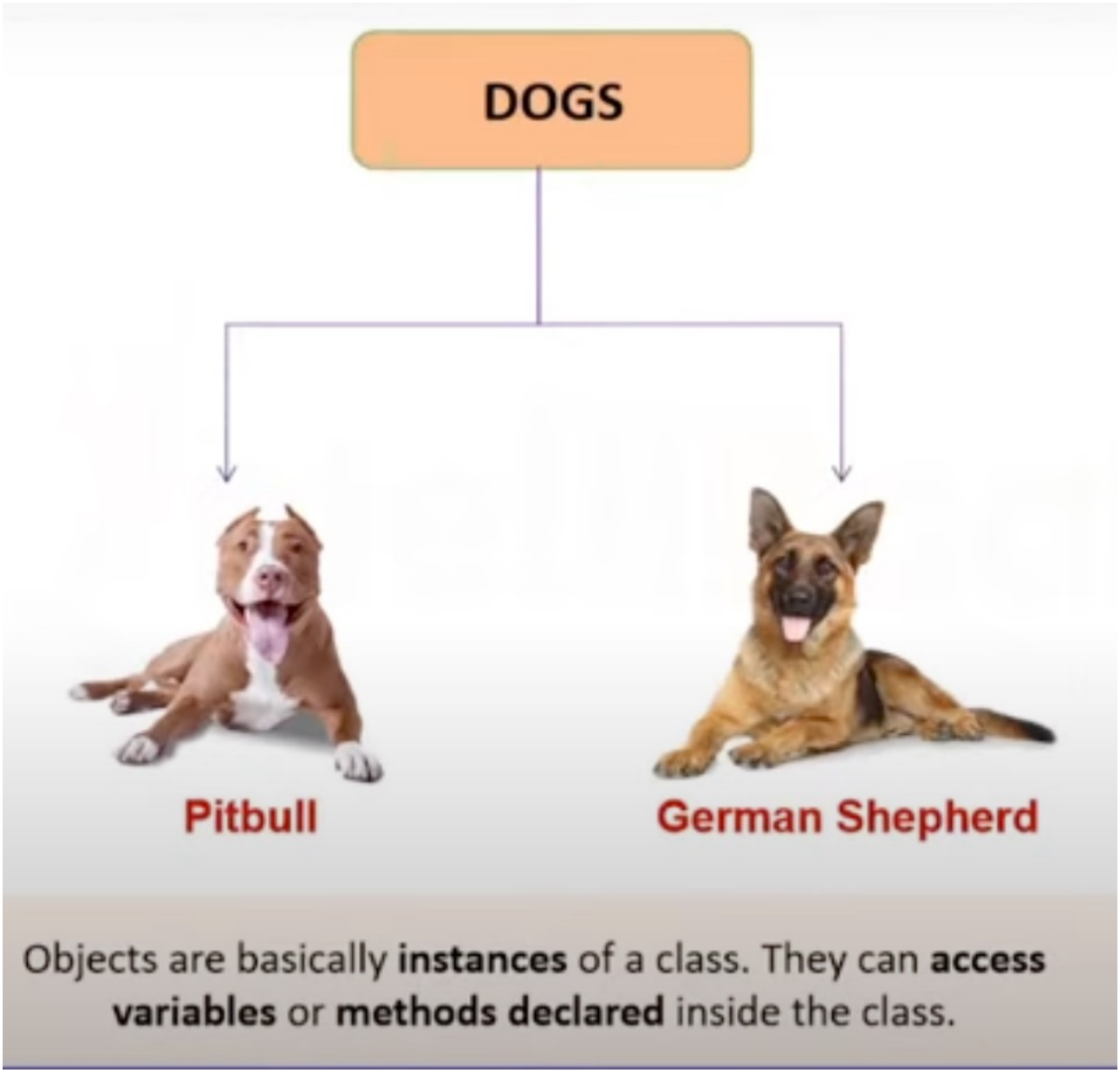
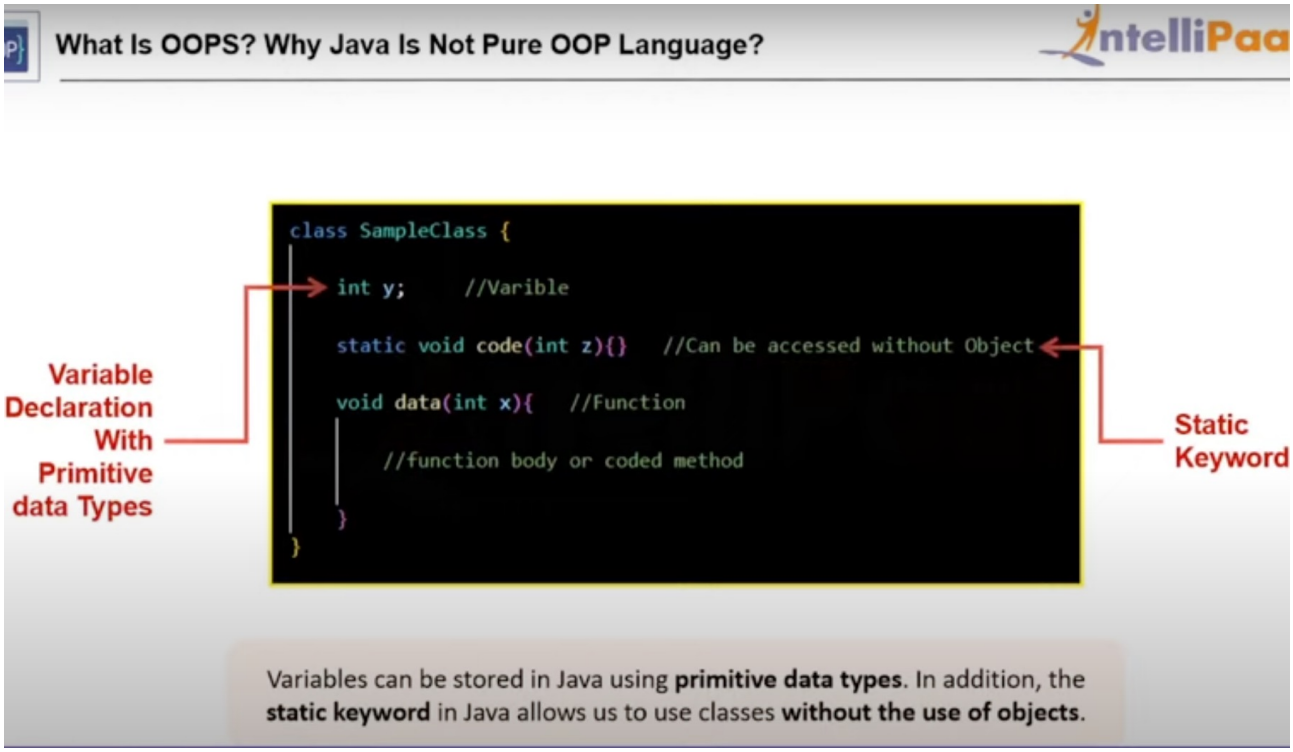
Oops concepts for interviews:






DOGS
Pitbull
German Shepherd
Objects are basically instances of a class. They can access variables or methods declared inside the class.


What Is OOPS? Why Java Is Not Pure OOP Language?
IntelliPaa
class SampleClass {
int y; //Varible
static void code(int z){} //Can be accessed without Object
void data(int x){ //Function
//function body or coded method
}
}
Variable Declaration With Primitive data Types
Static Keyword
Variables can be stored in Java using primitive data types. In addition, the static keyword in Java allows us to use classes without the use of objects.

Inheritance

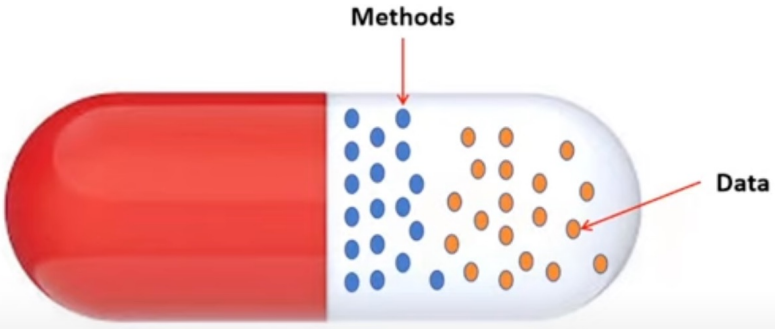
- With inheritance, a **derived class inherits the already existing class's features** (base class)
- Inheritance refers to using the **structure and behavior of a parent class in a subclass**
- Inheritance is **applied to classes**
- Inheritance can be of **single**, **hybrid**, **multiple**, **hierarchical**, **multipath**, and **multilevel types**.
- Inheritance supports **code reusability and reduces lines of code**

Polymorphism

- Polymorhism allows **class methods to exist in multiple forms**
- Polymorphism intends on **changing the behavior of parent class's method**
- Polymorphism is applied to **methods**
- Polymorphism has two types: **compile time and run time polymorphism**
- Polymorphism decides **which form of the function to be invoked**

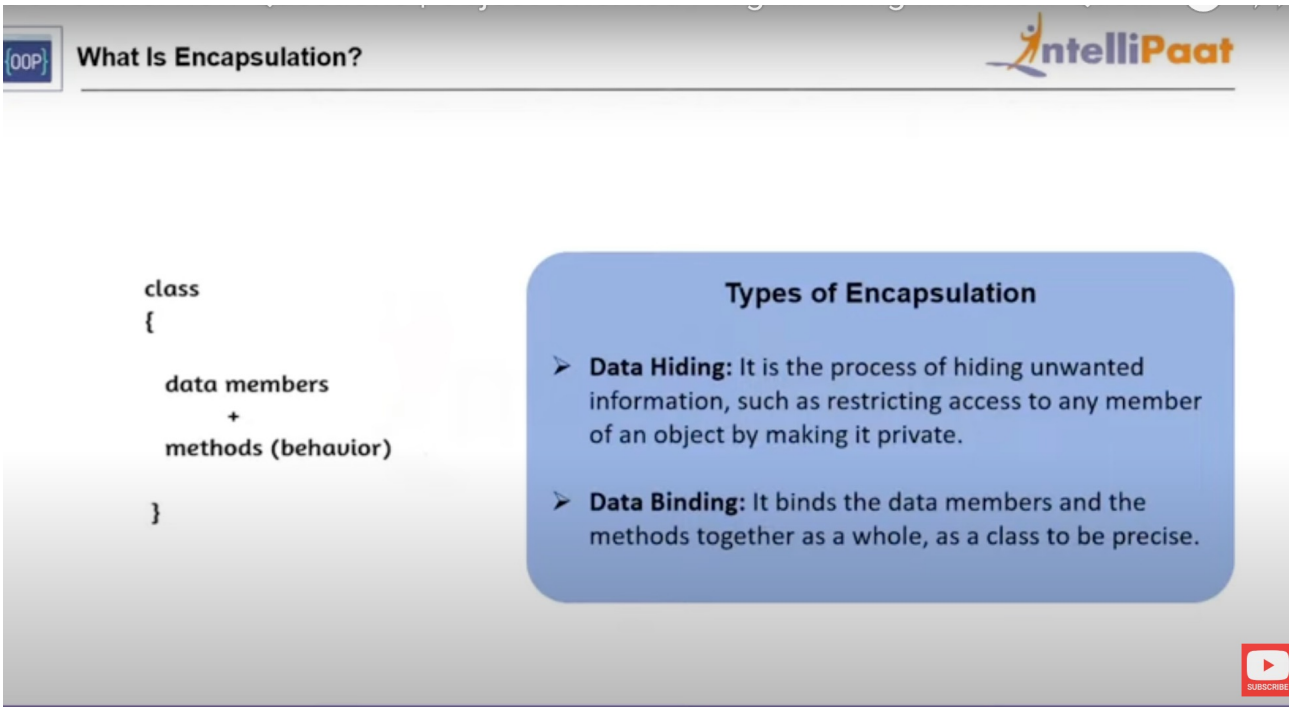
{OOP} What Is Encapsulation?

IntelliPaat

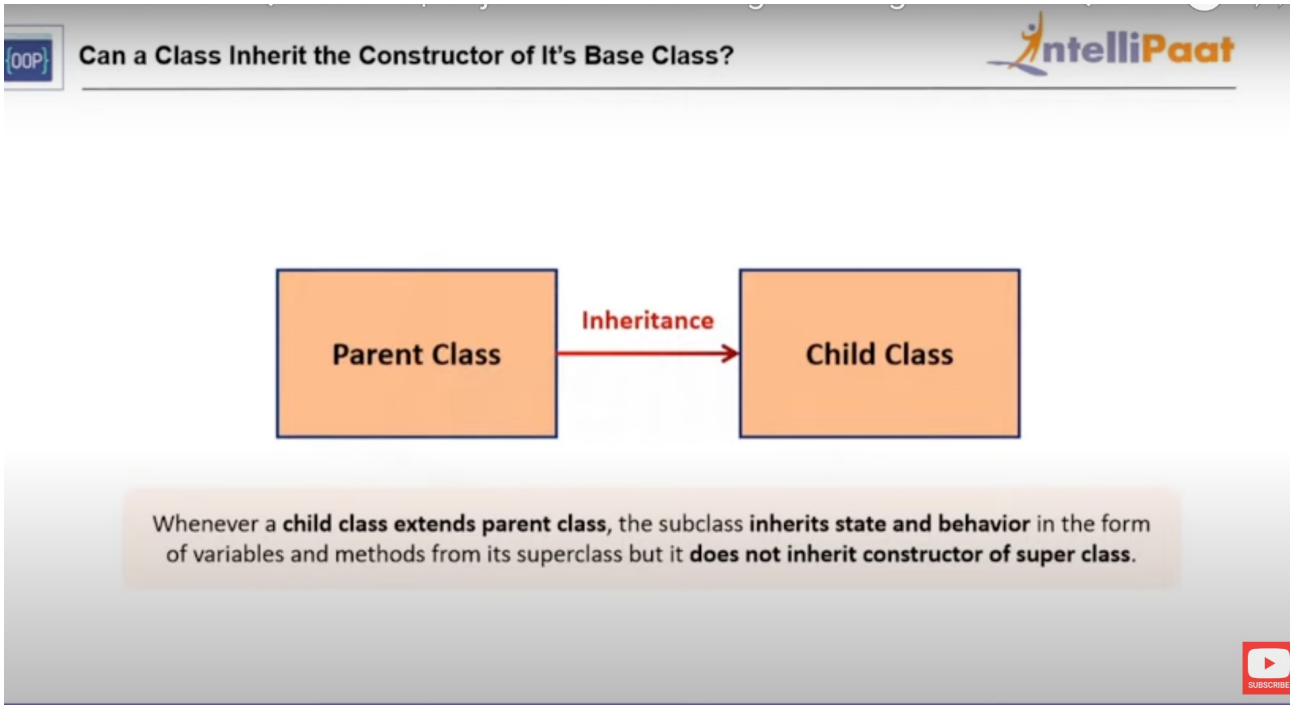


Class Binding Variables And Methods

Encapsulation is the **process of binding data members and methods** of a program together to do a specific job, **without revealing unnecessary details.**



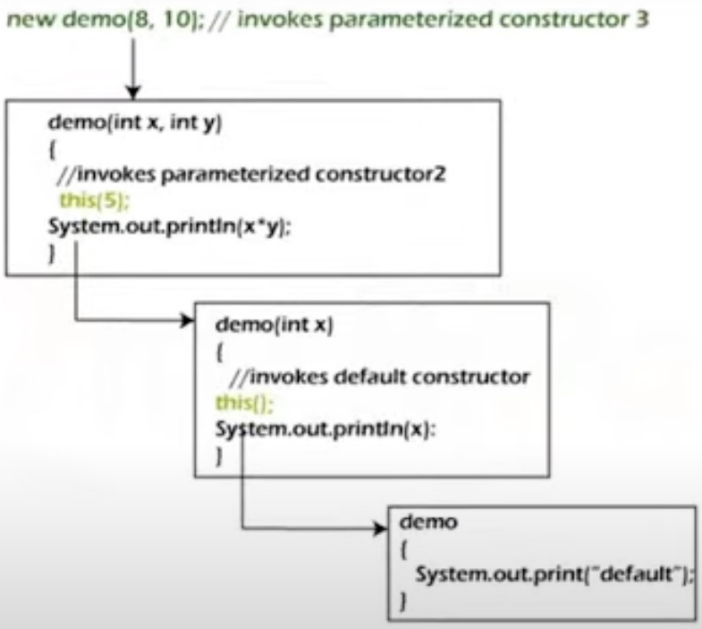
8. cAN we instantiate abstract class?



How Is an Abstract Class Different From an Interface?

- Both contains only the **method declaration** and not their **implementation**.
- When an interface is **implemented**, the **subclass must define all its methods** and provide its **implementation**.
- 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.

What Is Constructor Chaining?



Constructor chaining is a **sequence of invoking constructors of the same class** upon initializing an object.

What Is Singleton Class?

IntelliPaat

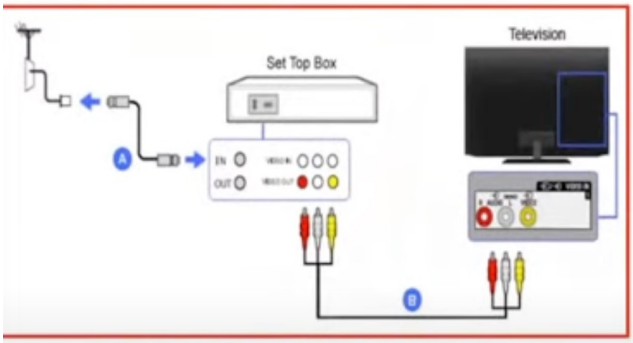
Singleton class is a class that can have **only one object at a time**. After the first instantiation, the **new object also points to the first instance**.

Singleton class is capable of **implementing interfaces, inheriting from other classes**, and **allowing inheritance**.

Static class on the other hand, **cannot inherit its instance members** which makes singleton classes more adaptable.

Abstraction Vs Encapsulation

IntelliPaa



Hiding Details of Television Component Circuitry: Encapsulation

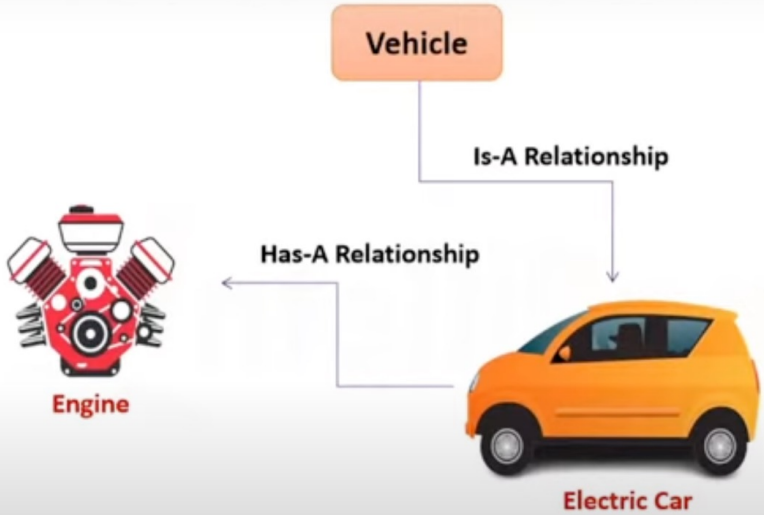
Hiding Everything Except TV Screen And Control Panel: Abstraction

Abstraction is about **expressing external simplicity** and **Encapsulation** is about **hiding internal complexity**.

What Is Constructor & Destructor? Can We Overload Them

- **Constructor:** It is a block of code similar to the method. A constructor will be invoked when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory space.
- **Destructor:** It is the last function that is called before an object is destroyed.

What Is Composition?



The composition is a way to design or implement the **has-a relationship**. The has-a relationship is used to implement an **instance variable that refers to another object**.

Copy Constructor Vs Assignment Operator

- The **copy constructor** and the **assignment operator (=)** both are used to initialize one object using another object.
- Copy constructor allocates **separate memory** to both objects i.e. existing object and newly created object
- Assignment operator **does not allocate new memory** for the **newly created object**

What Is Exception Handling?

BC — Exception is an event that occurs during the execution of a program at runtime and causes the execution to halt.

BD — Exception primarily occurs where the user wishes to do something that the program does not support

BE — Exceptions can be handled in the program ahead of time, preventing the execution from stopping.

BF — Try-catch is the most common method used for handling exceptions in the program.

What Is Coupling? Why It's Important?

- In programming, **separation of concerns** is known as **coupling**. It means that an object cannot change or modify the state or behavior of other objects directly.
- **Loosely Coupled Objects:** These objects are independent of one another and do not directly modify the state of other objects.
- **Tightly Coupled Objects:** Objects that depend on other objects and can modify the states of other objects are called tightly coupled.

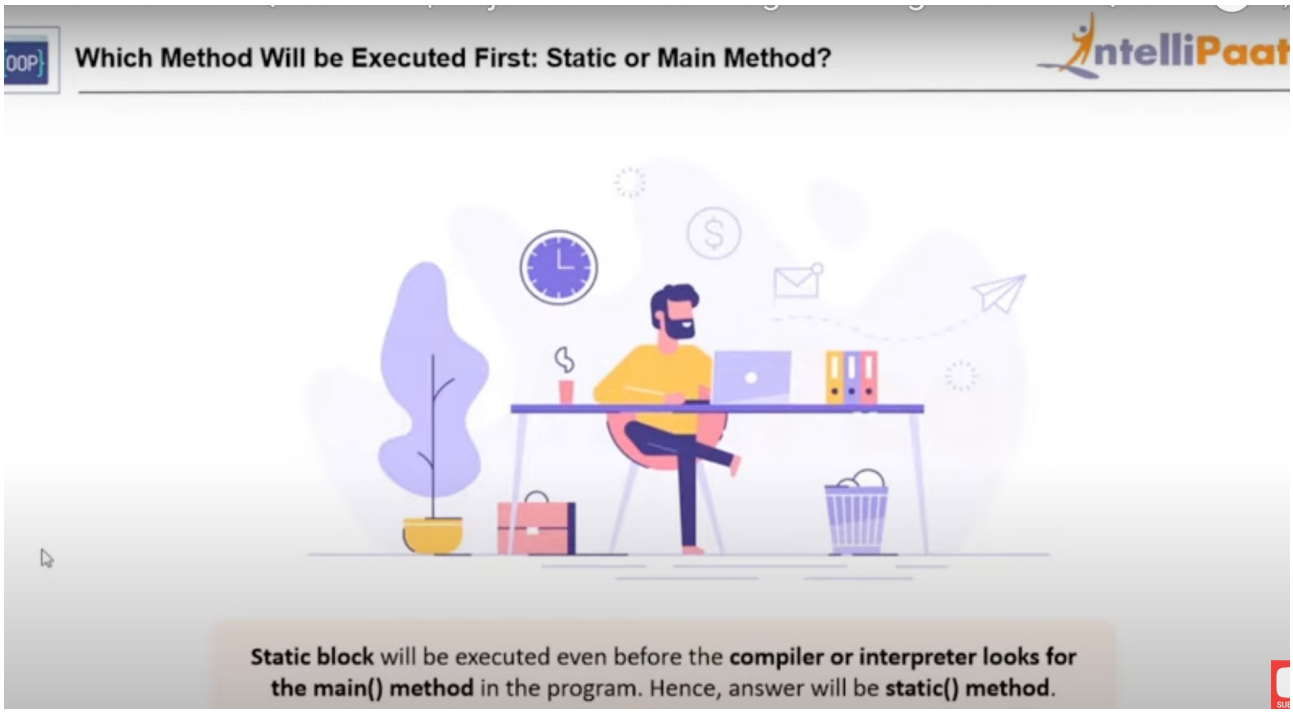
Explain Overriding And Overloading

Method Overloading is a concept in OOP which employs two or more methods in a class with the same name but different method signatures.

If a method with the same method signature is presented in both child and parent class is known as method overriding.

Can You Override the Constructor?

- You cannot **override the constructor** because it **looks like a method, but it is not**. It **lacks a return type and shares the same name as the class**.
- If you treat constructor as a method and write a **super class's constructor** in the **sub-class compiler** expecting a return type, it will generate a **compile-time error**.



I'd be happy to answer your advanced OOPs interview questions! Here are in-depth answers to each:

1. What is a class?

- A class is a blueprint or prototype that defines the properties (attributes) and behaviors (methods) of objects. It acts as a template for creating objects with similar characteristics and functionalities.

2. What is an object?

- An object is an instance of a class. It holds specific values for the properties defined in the class and exhibits the behaviors as defined by the class methods. Think of it as a real-world entity with attributes and actions.

3. What is encapsulation?

- Encapsulation refers to bundling data (attributes) and methods that operate on that data within a single unit (class). It promotes data hiding and protects data integrity by controlling access to attributes and methods using access specifiers (public, private, protected).

4. What is Polymorphism?

- Polymorphism allows objects of different classes to respond to the same method call in different ways. It provides flexibility and code reusability. There are two main types:
    - **Compile-time polymorphism:** Achieved through function overloading (multiple functions with same name but different parameter lists) and operator overloading (customization of operator behavior for user-defined types).
    - **Runtime polymorphism:** Achieved through virtual functions and inheritance. A virtual function allows subclasses to override its behavior, enabling dynamic method dispatch at runtime based on the object's actual type.

5. How does C++ support Polymorphism?

- C++ supports both compile-time and runtime polymorphism. Function and operator overloading provide compile-time polymorphism. Virtual functions and inheritance enable runtime polymorphism through dynamic method dispatch using the virtual table mechanism.

6. What is meant by Inheritance?

- Inheritance allows creating new classes (subclasses) that inherit properties and behaviors from existing classes (superclasses). This promotes code reusability, maintainability, and hierarchical organization of classes.

7. What is Abstraction?

- Abstraction focuses on exposing essential details while hiding internal complexities. It allows users to interact with objects without knowing their underlying implementation. This simplifies code understanding and usage.

8. How much memory does a class occupy?

- A class itself doesn't occupy memory. Objects created from a class allocate memory to store their specific attribute values. The amount of memory depends on the number and types of attributes defined in the class.

9. Is it always necessary to create objects from a class?

- No, not always. Some classes might contain static methods that don't operate on object data and can be called directly without creating an object. Additionally, abstract classes cannot be instantiated directly but serve as base classes for inheritance.

10. What is a constructor?

- A constructor is a special method with the same name as the class that gets automatically called when an object is created. It's used to initialize the object's attributes with appropriate values.

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

- C++ supports default constructors (no arguments), parameterized constructors (accept arguments to initialize attributes), copy constructors (create a new object by copying from an existing one), and destructor (called when an object is destroyed to release resources).

12. What is a copy constructor?

- A copy constructor is a special constructor that takes an existing object as an argument and creates a new object as a

copy of the original. It prevents shallow copying by ensuring deep copying of object members.

13. What is a destructor?

- A destructor is a special method with the same name as the class preceded by a tilde (~). It gets automatically called when an object goes out of scope or is explicitly deleted. It's used to release resources acquired by the object during its lifetime (e.g., closing files, deallocating memory).

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

- **Structure:**
  - Primarily holds data (attributes) but doesn't define methods by default.
  - Members are public by default.
  - Cannot inherit from other structures.
  - Faster to create and access members due to simpler memory layout.
- **Class:**
  - Can contain both data and methods.
  - Members can have different access specifiers (public, private, protected).
  - Supports inheritance.
  - More flexible and powerful but slightly slower due to potential virtual function table overhead.

15. Explain Inheritance with an example?

- Example: Consider a class `Vehicle` with attributes like `color`, `make`, and a method `startEngine`. You can create subclasses like `Car` and `Truck` that inherit these attributes and method, and add their specific attributes and methods (e

Advanced OOPs Concepts Explained:

16. Inheritance:

Inheritance allows creating new classes (subclasses) that inherit properties and behaviors from existing classes (superclasses). Imagine a hierarchy:

- Superclass: `Animal` (attributes: name, weight, methods: eat(), sleep())
- Subclasses: `Dog` (inherits from `Animal`, adds bark() method), `Bird` (inherits from `Animal`, adds fly() method)

Benefits: code reusability, maintainability, hierarchical organization.

17. Limitations of Inheritance:

- **Rigidity:** Subclasses tightly coupled to superclass, changes in superclass can affect subclasses unexpectedly.
- **Complexity:** Deep inheritance hierarchies can become hard to understand and maintain.
- **Diamond Problem:** Multiple inheritance can lead to ambiguity when resolving inherited members from multiple parent classes.

18. Types of Inheritance:

- **Single Inheritance:** A subclass inherits from one superclass. (e.g., `Dog` inherits from `Animal`)
- **Multiple Inheritance:** A subclass inherits from multiple superclasses. (not recommended due to diamond problem)
- **Multilevel Inheritance:** A subclass inherits from another subclass, which in turn inherits from another, forming a chain.
- **Hierarchical Inheritance:** Multiple subclasses inherit from the same superclass.

19. Subclass: A class that inherits from another class. (e.g., `Dog` is a subclass of `Animal`)

20. Superclass: A class from which other classes inherit. (e.g., `Animal` is a superclass of `Dog`)

21. Interface: A blueprint defining methods that a class must implement, focusing on what a class can do, not how. Promotes decoupling and loose coupling. (e.g., an `Eatable` interface with an `eat()` method)

22. Static Polymorphism: Achieved through function/operator overloading at compile time based on argument types. (e.g., multiple `print()` functions for different data types)

23. Dynamic Polymorphism: Achieved through virtual functions and inheritance. Method call resolved at runtime based on the object's actual type. (e.g., an `Animal` object calling `makeSound()` delegates to the appropriate subclass like `Dog`'s `bark()` or `Bird`'s `chirp()`)

24. Overloading vs. Overriding:

- **Overloading:** Multiple functions with the same name but different parameter lists within the same class.
- **Overriding:** Redefining a method inherited from a superclass in a subclass to provide specific behavior.

25. Data Abstraction: achieved through encapsulation, hiding implementation details while exposing essential functionalities through methods. Users interact with objects without knowing their internal workings.

26. Abstract Class: A class with at least one pure virtual function (without implementation) and cannot be instantiated directly. Serves as a base for inheritance, forcing subclasses to implement the pure virtual functions.

27. Abstract Class vs. Interface:

- **Abstract Class:** Can have both abstract and concrete methods, providing partial implementation.
- **Interface:** Only declares pure virtual functions, no implementation, enforces stricter contracts.

28. Access Specifiers:

- **public:** Members accessible from anywhere.
- **private:** Members accessible only within the class.
- **protected:** Members accessible within the class and its subclasses.

29. Exception: An event that disrupts the normal flow of a program.

30. Exception Handling: Mechanism to handle exceptions gracefully, preventing program crashes. Includes try-catch blocks.

31. Garbage Collection: Automatic memory management in OOP languages, reclaiming memory of unused objects.

32. Java Application without OOPs:

- Technically possible using procedural programming techniques, but impractical and discouraged. OOPs concepts form the foundation of modern Java development.

Remember, these are brief explanations. Feel free to ask further questions for deeper understanding!

OOPs Coding Problems

1. What is the output of the below code?

Explain

```
#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.

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

Explain

```
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.

3. Predict the output?

Explain

```
#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.

4. What will be the output in below code?

Explain

```
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.

5. Predict the output?

Explain

```
#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 DerivedBaseClass2 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.

6. What is the output of the below program?

Explain

```
#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:

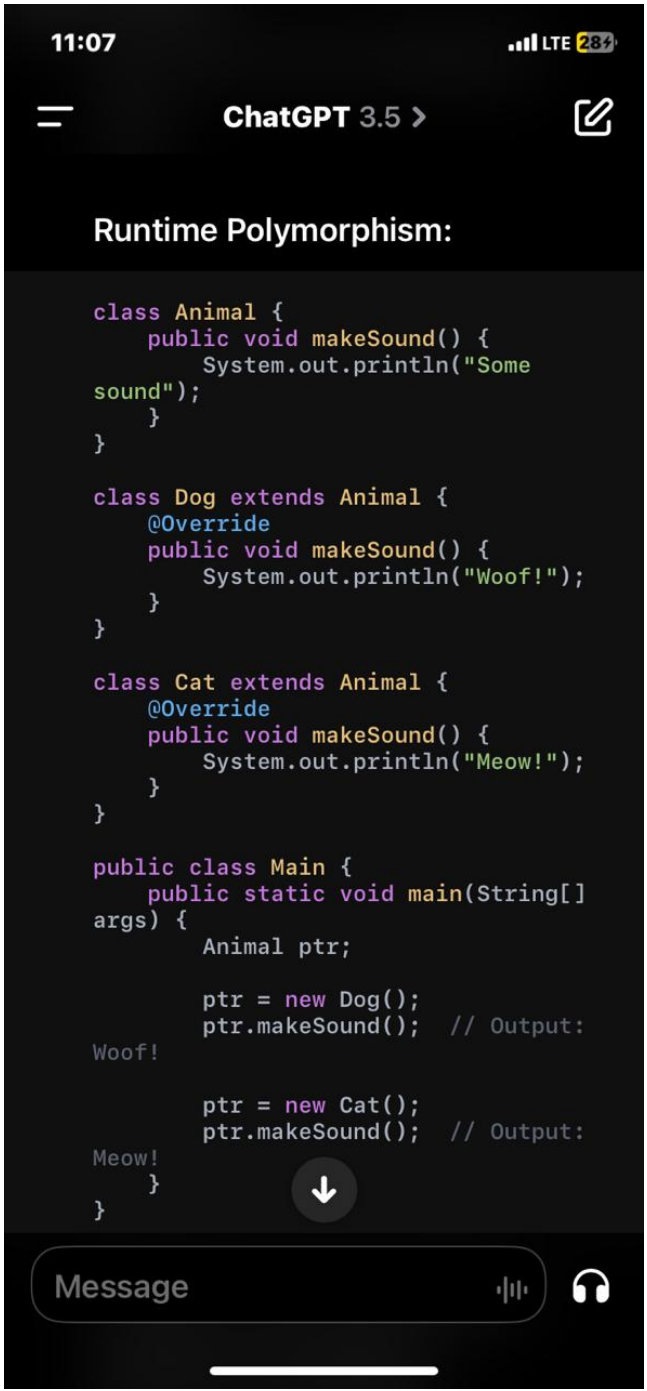
```
Inside B
```

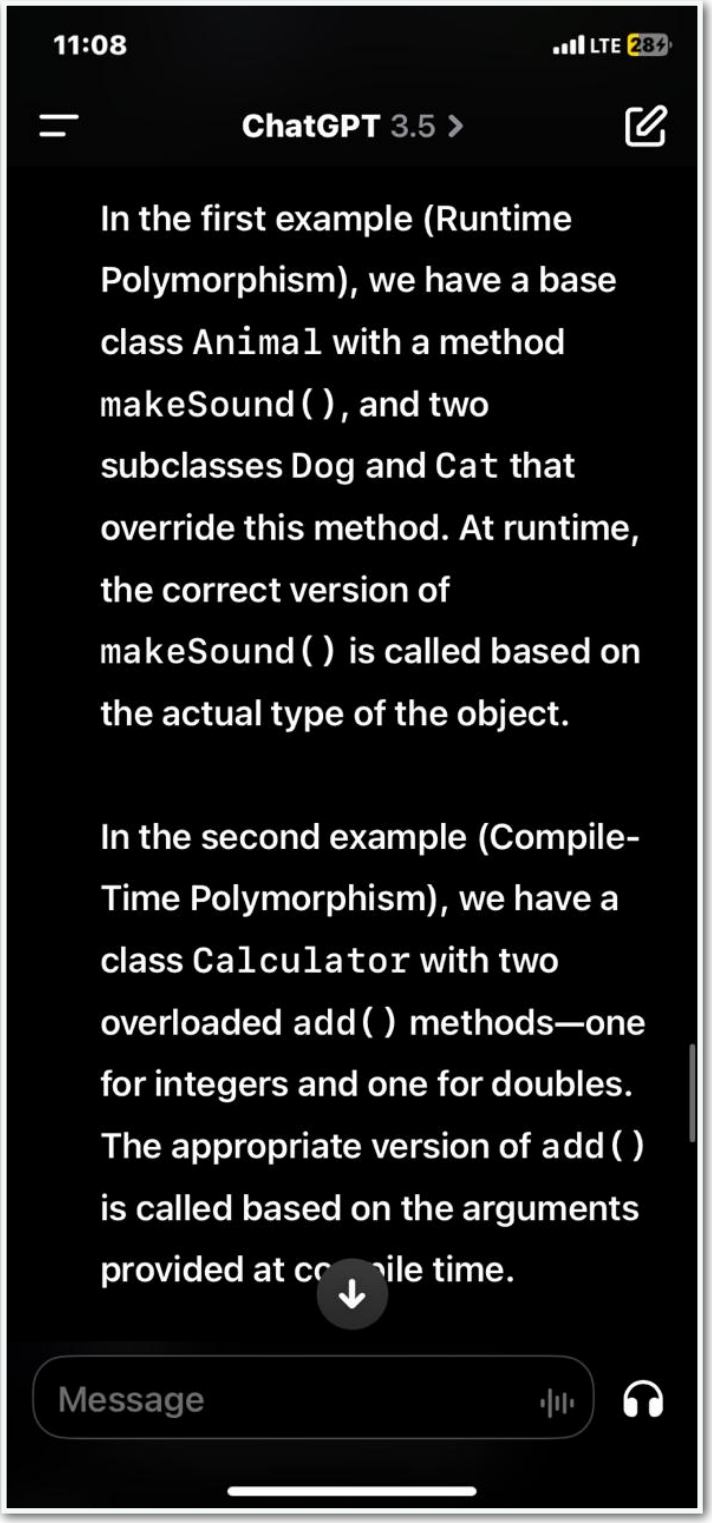
Reason:

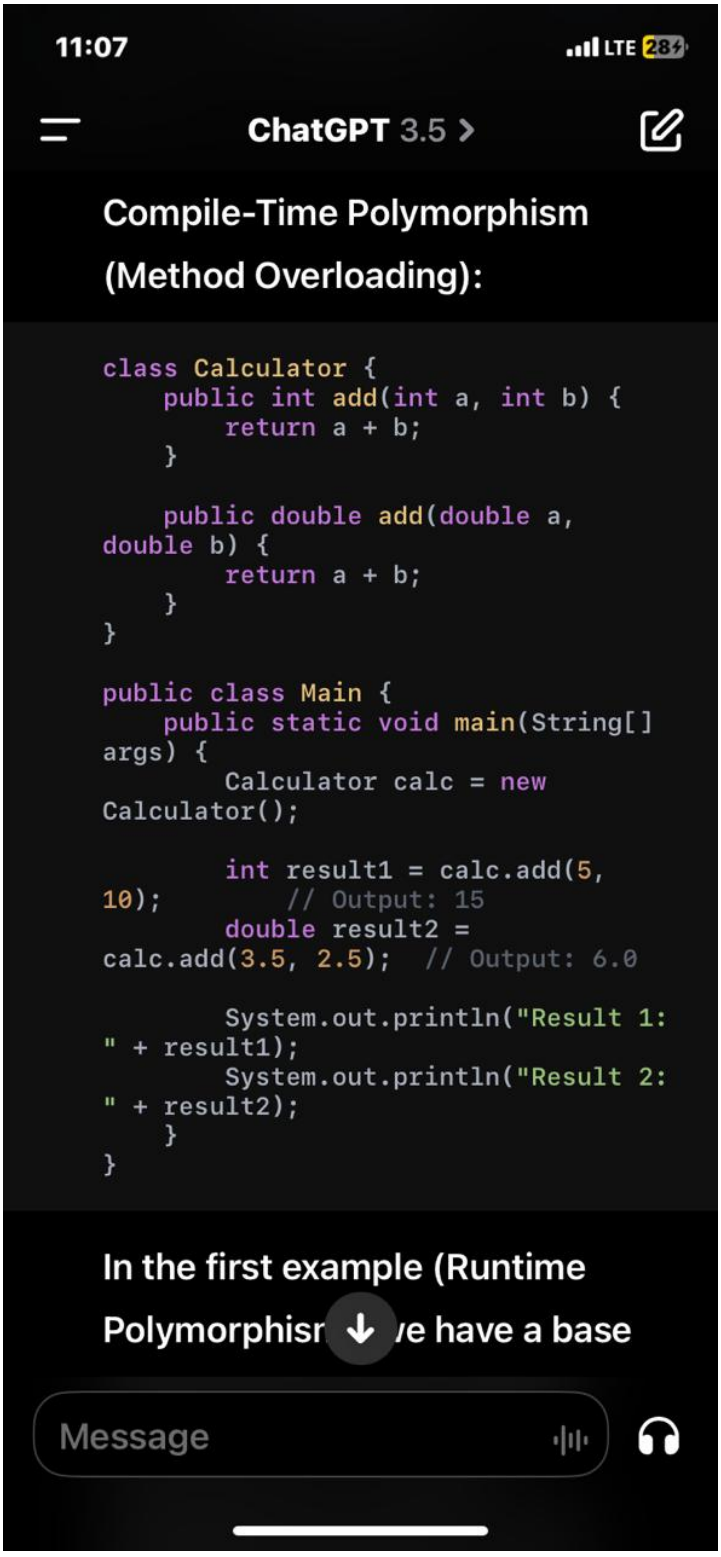
The above program implements a Multi-level hierarchy. So the program is linearly searched up until a matching function is found. Here, it is present in both classes A and B. So class B's print() method is called.

Useful Resource

[Features of OOPS]




11:08
LTE
ChatGPT 3.5 >
In the first example (Runtime Polymorphism), we have a base class Animal with a method makeSound(), and two subclasses Dog and Cat that override this method. At runtime, the correct version of makeSound() is called based on the actual type of the object.
In the second example (Compile-Time Polymorphism), we have a class Calculator with two overloaded add() methods—one for integers and one for doubles. The appropriate version of add() is called based on the arguments provided at co ile time.
Message


11:07
LTE
ChatGPT 3.5 >
Compile-Time Polymorphism (Method Overloading):
class Calculator {
public int add(int a, int b) {
return a + b;
}
public double add(double a, double b) {
return a + b;
}
}
public class Main {
public static void main(String[] args) {
Calculator calc = new Calculator();
int result1 = calc.add(5, 10); // Output: 15
double result2 = calc.add(3.5, 2.5); // Output: 6.0
System.out.println("Result 1: " + result1);
System.out.println("Result 2: " + result2);
}
}
In the first example (Runtime Polymorphism) we have a base
Message