Object Oriented Programming

OOPs

Table of Contents

OOPs (Object Oriented Programming)

Object-Oriented Programming is a programming paradigm that allows you to m
world entities as objects and define their behaviour through methods and attri
promotes modularity, reusability, and maintainability of code.

How OOPs is better?

1. **Modularity**: OOP promotes modular design, allowing you to break compl
   into manageable parts (objects) with distinct responsibilities.
2. **Reusability**: With features like inheritance, you can create new classes by
   existing ones, reducing redundant code and improving code reuse.
3. **Encapsulation**: Encapsulation hides the internal details of objects, prever
   unintended interference and making code more robust and secure.
4. **Abstraction**: OOP allows you to model real-world concepts abstractly, fo
   what objects do rather than how they do it.
5. **Flexibility**: Polymorphism enables you to use different objects interchang
   fostering adaptable and flexible code.
6. **Scalability**: OOP promotes scalability as the codebase grows. You can ext
   classes or create new ones without disrupting the existing code.
7. **Maintenance**: Changes in one part of the codebase have limited impact
   leading to easier maintenance and updates.
8. **Collaboration**: Teams can work concurrently on different classes or modu
   interfering with each other's work.
9. **Real-World Modelling**: OOP models real-world entities and their relation
   the codebase more intuitive and closely mirroring the problem domain.
10. **Code Understandability**: Well-designed OOP code with clear class hierarc
    meaningful names enhances code readability and understandability.

Overall, OOP provides a structured approach that improves code organizati
maintainability, and collaboration among developers.

Main features of OOPs

1. **Encapsulation**: Bundling data (attributes) and methods that operate on t
   single unit (object) while hiding internal details.
   Eg: In a smartphone, there are various components such as the processo
   camera, and battery. These components are encapsulated within the de
   shell, which serves as a protective barrier. Users interact with the smartp
   a limited set of well-defined interfaces, such as the touchscreen, buttons

2. **Abstraction**: Abstraction is a OOPs concept which "shows" only essential
   and "hides" unnecessary information from the outside.
   Eg: Abstraction is like using a TV remote without knowing its inner worki
   interact with its buttons (interface) to control the TV, without needing to
   the technical details inside.

3. **Inheritance**: Creating new classes by inheriting attributes and methods fr
   ones, promoting code reuse and hierarchy.
   Eg: Think of a vehicle hierarchy: all vehicles share common traits like hav
   and engines. Inheritance is like having a "Vehicle" class as the base, and
   subclasses like "Car," "Bike," and "Truck," inheriting the basic attributes f
   "Vehicle" class while adding specific features for each type of vehicle.

4. **Polymorphism**: Treating different objects through a common interface, a
   flexibility and dynamic behaviour in code.
   Eg: Imagine a music player: different types of devices like phones, tablet
   can all play music. Polymorphism allows you to control the music playba
   same play, pause, and stop buttons on these devices, even though the u
   mechanisms are different, making it easy to interact with different objec
   way.
   Eg: A person at the same time can have different characteristics. Like a r
   same time is a father, a husband, and an employee.

Basic Terminologies:-

1. **Class:** A class is a building block of Object Oriented Programs. It is a user type that contains the data members and member functions that operat members. It is like a blueprint or template of objects having common pr methods.

2. **Object**: An object refers to the instance of the class, which contains the i members and behaviors defined in the class template. In the real world, an actual entity to which a user interacts, whereas class is just the bluep object.

```
# Class keyword is used to create a class
class Car:
    def __init__(self, make, model, year):
        #self is used to access the attributes and methods of the class
        self.make = make
        self.model = model    #atributes
        self.year = year

    def start_engine(self):   #methods
        return f"{self.make} {self.model}'s engine started."

# Creating objects (instances) of the Car class
car1 = Car("Toyota", "Camry", 2021)
car2 = Car("Tesla", "Model S", 2022)
```

3. **Attributes and Methods**: Attributes are variables that hold data specific t while methods are functions that define the behaviour of the object.

4. **Constructor:** A constructor is a block of code that initializes the newly cre constructor resembles an instance method but it's not a method as it do return type.
   The __init__ method is a special method in Python classes that gets calle automatically when an object is created from the class. It is used to initia object's attributes with the values provided during object creation.

```python
# Example: Default Arguments in the Car class constructor
#__init__ is constructor
class Car:
    def __init__(self, make="Unknown", model="Unknown", year=2020):
        self.make = make
        self.model = model
        self.year = year

car1 = Car("Toyota", "Camry", 2021)
car2 = Car()  # Using default values for attributes
```

5. **Destructors:** Destructors are also special methods. But destructors free u resources and memory occupied by an object. Destructors are automati when an object is being destroyed.
   Class destructors are useful for performing clean-up tasks, such as releas like file handles or network connections, before an object is removed fro

```python
# Example: Using Class destructors(__del__) for resource release in the Car
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.file_handle = open(f"{make}_{model}_data.txt", "w")

    def __del__(self):
        self.file_handle.close()
        print(f"{self.make} {self.model} object is being destroyed.")

car = Car("Toyota", "Camry")
# Assume other operations with the car object and file handling.
del car  # File handle will be closed, and the object is destroyed.
```

Encapsulation

Encapsulation refers to binding the data and the code that works on that toget
unit. Class binds variables and methods to perform some task, hence implemer
encapsulation.

Access specifiers or **access modifiers** are keywords that determine the accessib
methods, classes, etc in OOPs. These access specifiers allow the implementatio
encapsulation. Three type of access modifiers are:-

1. Public: All the class members declared under the public specifier will be
everyone.
2. Private: The class members declared as private can be accessed only by
functions inside the class. They are not allowed to be accessed directly b
or function outside the class. Only the member functions or the friend fu
available in python) are allowed to access the private data members of t
3. Protected: The protected access modifier is similar to the private access
the sense that it can't be accessed outside of its class, however they can
by any subclass (derived class) of that class.

Name	Accessibility from own class	Accessibility from derived class	Accessi
Public	Yes	Yes	
Private	Yes	No	
Protected	Yes	Yes	

Access Modifiers in Python

Python uses access modifiers to control the visibility of attributes and methods
class. The three access modifiers are:

- Public: No underscore before the attribute/method name. Accessible fr

The default access modifier for class members (attributes and methods) is pub

```
class Employee:
    def __init__(self, name, salary):
        self.name = name                # Public attribute
        self._salary = salary           # Protected attribute
        self.__bonus = 1000             # Private attribute

    def get_bonus(self):
        return self.__bonus

    def _calculate_total_salary(self):
        return self._salary + self.__bonus

employee = Employee("Alice", 50000)

# Accessing public and protected attributes directly
print(employee.name)       # Output: "Alice"
print(employee._salary)    # Output: 50000

# Attempting to access the private '__bonus' attribute directly
# This will raise an error: AttributeError: 'Employee' object has no attribu
print(employee.__bonus)   # Output: Attribute error

# Accessing the private '__bonus' attribute using the getter method
print(employee.get_bonus())  # Output: 1000

# Accessing the protected attribute and private attribute through a method
print(employee._calculate_total_salary())  # Output: 51000
```

Private methods: cannot be accessed directly, but public methods of same fun
this.

```
class MyClass:
    def __init__(self):
        self.__private_var = 42

    def public_method(self):
        print("This is a public method.")
        self.__private_method()  # A public method can call a private method

    def __private_method(self):
        print("This is a private method.")

# Create an instance of the class
```

Abstraction

Abstraction is a fundamental concept used to simplify complex systems by focu essential details while hiding the unnecessary complexities. Abstraction allows software that is easier to understand, maintain, and scale. At its core, abstracti

1. **Hiding Implementation Details:** Abstraction allows us to hide the complex workings of a system or an object. This is crucial because it reduces com allows us to work with high-level concepts.
2. **Exposing Only Necessary Information**: Abstraction exposes only the relev properties, and behaviors of an object, making it easier for developers to it.

Ways to achieve abstraction in python

- Use abstract classes to define a blueprint for group of related classes.
- Mark attributes and methods as private by prefixing their names with a d underscore __. This indicates they are intended for internal use within th
- Use getter and setter methods to provide controlled access to private at
- Leverage Python modules and libraries to abstract complex functionality

Eg1: **Getter and Setter Methods:** To provide controlled access to private attribu use getter and setter methods. Getter methods retrieve the value of private at setter methods modify their values.

```
class MyClass:
    def __init__(self):
        self.__private_var = 42

    def get_private_var(self):
        return self.__private_var

    def set_private_var(self, value):
```

For example, the math module provides a range of mathematical functions and abstracting the low-level implementation details.

Abstract Classes

An **abstract class** in Python is a class that cannot be instantiated, meaning you c objects directly from it. Instead, abstract classes serve as blueprints for other c typically contain one or more **abstract methods** that should be implemented by

Abstract classes and methods help ensure that specific behaviors are implemen consistently across different classes in your program. Other languages use inte functionality (Interfaces specify the method signatures (names, parameters, re that classes must provide implementations for).

Defining Abstract Classes

To define an abstract class in Python, you can use the abc module (Abstract Ba Here's how you define an abstract class:

```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass
```

In this example, MyAbstractClass is an abstract class, and my_abstract_method method that must be implemented by any subclass.

Sub classing and Implementing Abstract Methods

When you create a subclass of an abstract class, you must provide implementa abstract methods. Failure to do so will result in a TypeError.

```
class MyConcreteClass(MyAbstractClass):
    def my_abstract_method(self):
        return "Implemented abstract method in MyConcreteClass"
```

Inheritance

Inheritance allows one class to inherit properties (attributes and methods) fror class. The class that inherits is called the subclass or derived class, and the class inherits is known as the superclass or base class.

```
#Base class
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        return "Some generic sound"

    def get_species(self):
        print("My species is",self.species)

#Subclass
class Dog(Animal):
    def __init__(self, breed):
        super().__init__("Dog")
        self.breed = breed

    def make_sound(self):
        return "Woof!"

# Creating objects of the classes
animal_obj = Animal("Unknown")
dog_obj = Dog("Labrador")

print(animal_obj.species)  # Output: "Unknown"
print(dog_obj.species)     # Output: "Dog"
print(dog_obj.make_sound())  # Output: "Woof!"

#subclass obj can use methods of base class too
dog_obj.get_species() # My species is Dog
```

In the above example, the Dog class inherits from the Animal class. The Dog cla attribute breed and overrides the make_sound method with its implementatio

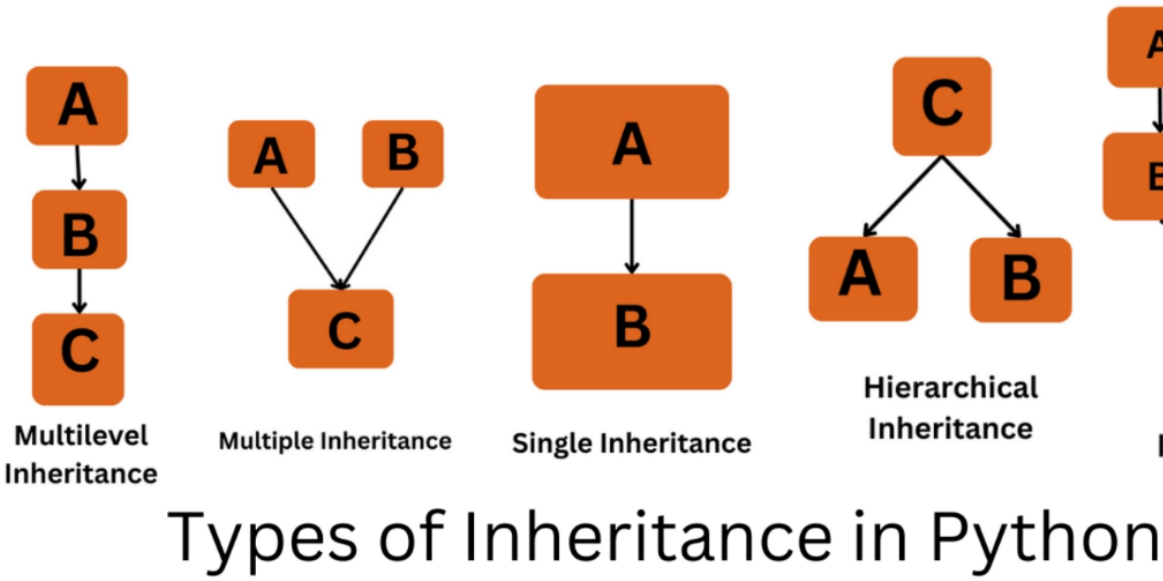
```python
# Example: Using super() to call the superclass method
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        return "Some generic sound"

class Dog(Animal):
    def __init__(self, breed):
        super().__init__("Dog")
        self.breed = breed

    def make_sound(self):
        generic_sound = super().make_sound()
        return f"{generic_sound} but also Woof!"

dog_obj = Dog("Labrador")
print(dog_obj.make_sound())  # Output: "Some generic sound but also Woof!"
```

Method Overriding

Method overriding occurs when a subclass provides its implementation for a m
already defined in the superclass. The subclass method with the same name as
superclass method will override the superclass method.

```python
# Example: Method Overriding in the Car class
class Vehicle:
    def start(self):
        return "Vehicle starting..."

class Car(Vehicle):
    def start(self):
        return "Car starting..."


vehicle_obj = Vehicle()
car_obj = Car()

print(vehicle_obj.start())  # Output: "Vehicle starting..."
```

Types of inheritance



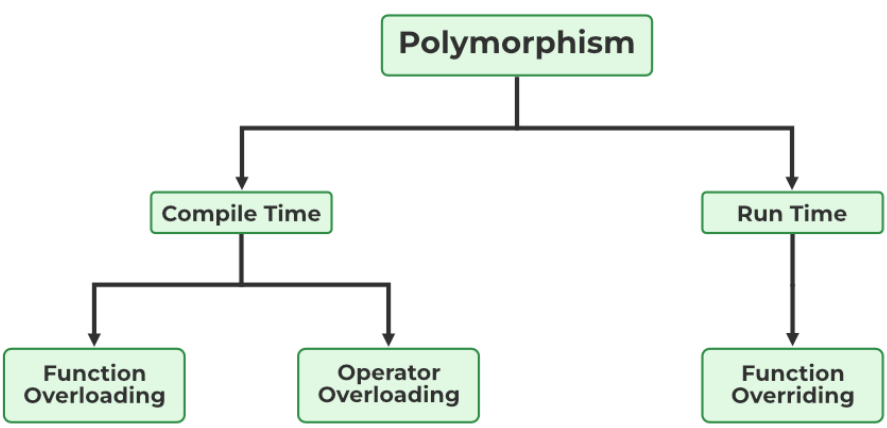
To implement multiple inheritance

```
# multiple inheritance

# Base class1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class2
class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)


# Driver's code
```

Polymorphism

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

- **Compile-Time Polymorphism**: Compile time polymorphism, also known a
  polymorphism or early binding is the type of polymorphism where the b
  call to its code is done at the compile time. Method overloading or opera
  overloading are examples of compile-time polymorphism.
- **Runtime Polymorphism**: Also known as dynamic polymorphism or late bi
  polymorphism is the type of polymorphism where the actual implementa
  function is determined during the runtime or execution. Method overrid
  example of this method.



Method overloading

Method overloading allows you to define multiple methods with the same nam
but with different parameters.

Python does not natively support method overloading as some other language
or C++).

```
class Calculator:
    def add(self, a, b=None):
        if b is None:
            return a
        else:
            return a + b

calc = Calculator()
result1 = calc.add(5)
result2 = calc.add(2, 3)

print(result1)  # Output: 5
print(result2)  # Output: 5
```

Cant do this(the real way of function overloading in other languages)

```
class Calculator:
    def add(self, a, b=None):
        return a+b

    def add(self,a,b,c):    #gets type error when tried to use
        return a+b+c
```

Operator overloading

Operator overloading in Python allows you to define custom behavior for stand
when applied to objects of your class. This is achieved by defining special meth
double underscores (e.g., __add__() for addition) in your class. These special m
also known as "**magic methods**" or "**dunder methods**" (short for "double unders

Special methods are enclosed in double underscores (__) at the beginning and
names. They provide a way to define specific behaviors for objects, making Pyt
more powerful and flexible

1. **The __str__ method** is used to provide a human-readable string represen
   object. It is called when you use the str() function or the print statement
   object.
2. **The __add__ method** allows you to define addition between objects of y
   called when you use the + operator with the objects.

```python
# Example: Special Methods in Python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

point1 = Point(1, 2)
point2 = Point(3, 4)

print(point1)               # Output: "Point(1, 2)"
print(point1 + point2)      # Output: "Point(4, 6)"
print(point1 == point2)     # Output: False
```

Eg2: Use of sub and mul methods

```python
# Example: Other Common Special Methods
class ComplexNumber:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, other):
        return ComplexNumber(self.real + other.real, self.imag + other.imag)

    def __sub__(self, other):
        return ComplexNumber(self.real - other.real, self.imag - other.imag)

    def __mul__(self, other):
        return ComplexNumber(self.real * other.real - self.imag * other.imag
                             self.real * other.imag + self.imag * other.real

    def __str__(self):
        return f"{self.real} + {self.imag}i"

num1 = ComplexNumber(2, 3)
```

Advance OOPs concept

Method Resolution Order (MRO's)

When a class inherits from multiple base classes, Python uses a method resolut
(MRO) to determine the order in which the base classes' methods are called.

```python
# Example: Method Resolution Order (MRO) in Multiple Inheritance

class A:
    def greet(self):
        return "Hello from A."

class B(A):
    def greet(self):
        return "Hello from B."

class C(A):
    def greet(self):
        return "Hello from C."

class D(B, C):
    pass

d = D()
print(d.greet())  # Output: "Hello from B."
```

In this example, the D class inherits from both B and C, and since B is listed first
inheritance chain, its method is called when we call the greet method on an ob
class.

Mixins

Mixins are a way to share functionalities among classes without using multiple
mixin is a class that is not intended to be instantiated but is designed to be mix
classes, enhancing their functionalities.

```python
# Example: Using Mixins in Python
class JSONMixin:
    def to_json(self):
        import json
        return json.dumps(self.__dict__)

class XMLMixin:
    def to_xml(self):
        xml_str = f"<{self.__class__.__name__}>"
        for key, value in self.__dict__.items():
            xml_str += f"<{key}>{value}</{key}>"
        xml_str += f"</{self.__class__.__name__}>"
        return xml_str

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Employee(Person, JSONMixin, XMLMixin):
    def __init__(self, name, age, emp_id):
        super().__init__(name, age)
        self.emp_id = emp_id

employee = Employee("John", 30, "EMP123")

print(employee.to_json())
# Output: '{"name": "John", "age": 30, "emp_id": "EMP123"}'

print(employee.to_xml())
# Output: '<Employee><name>John</name><age>30</age><emp_id>EMP123</emp_id></
```

Decorator Functions

Decorators allow you to extend or modify the behavior of callable objects (fun
methods) without changing their actual code.

Decorators in Python are functions that modify the behavior of other functions

```python
# Decorator function
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

# Using the decorator
@my_decorator
def say_hello():
    print("Hello!")

# Calling the decorated function
say_hello()

# We get the output
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

Few built-in decorators:

1. @property:

   Use: To define a method that can be accessed like an attribute.

   Example Use Case: The @property decorator allows you to define a metho accessed like an attribute. The @value.setter decorator allows setting the v were an attribute.

```python
class MyClass:
    def __init__(self):
        self._value = None

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
```

2. @staticmethod:

Use: To define a static method within a class.

Example Use Case: Defining utility methods that don't require access to the attributes.

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

result = MathOperations.add(3, 5)
```

3. @classmethod:

Use: To define a class method within a class.

Example Use Case: Creating alternative constructors or methods that involv itself rather than instances. The @classmethod decorator is used to define These methods take the class itself as the first argument, allowing you to w level attributes.

```
class MyClass:
    class_variable = "I am a class variable"

    def __init__(self, instance_variable):
        self.instance_variable = instance_variable

    @classmethod
    def create_instance(cls, value):
        return cls(value)

obj = MyClass.create_instance("Hello")
```

4. @staticmethod (builtin function):

Use: To define standalone static functions.

Example Use Case: Defining utility functions that are related to a concept bu require an instance or class context. You can also use @staticmethod as a s decorator for functions outside of a class. This is not to be confused with th method decorator.

5. @classmethod (builtin function):

Use: To define standalone class methods.

Use Case: Similar to class methods within a class, but are outside of a class

```
@classmethod
def my_class_method(cls):
    print(f"I am a class method in class {cls}")

class MyClass:
    my_class_method()
```

Static Methods

Static methods are methods that belong to a class rather than an instance of th
are shared among all instances of the class and can be called on the class itself
creating an instance.

The @staticmethod decorator is used for defining static methods in a class. Th
don't have access to the instance or its attributes.

```
class MyClass:
    class_variable = "I am a class variable"

    def __init__(self, value):
        self.instance_variable = value

    @staticmethod
    def static_method():         #do not have access to class_variable(do no
        print("This is a static method")

    @staticmethod
    def add(x,y):
        print(x+y)

    @classmethod
    def class_method(cls):
        print(f"This is a class method. Class variable: {cls.class_variable}

MyClass.static_method() # Calling static method without an instance(output:This
MyClass.add(2,3) (output: 5)
```

Interview Questions

Q1. How much memory does a class occupy?

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

Q2. Why OOPs is important?

Object-Oriented Programming (OOP) is vital for software development modularity, enhancing code organization. Through encapsulation, OOP ensur hiding internal complexities. Additionally, features like inheritance facilitat fostering maintainability and scalability. Overall, OOP provides a structured paradigm for building robust, adaptable, and secure software systems.

Q3. Are classes and structures same?

Classes and structures share commonalities in object-oriented programming, in fundamental aspects. Classes, typically used for complex data structures, all support inheritance, and default to private member accessibility. They are re often allocated on the heap, and are designed for encapsulating behavior. On t structures are often employed for lightweight data containers with immut inheritance support, and defaulting to public member accessibility. They ar usually allocated on the stack, and are suitable for simple data representation

Q4. What are pure OOPs languages, is Java a pure Object-Oriented Programmin

Pure Object-Oriented Programming (OOP) languages are those where everythir an object, and all operations are performed using objects. Examples include Ruby. Java, while often considered object-oriented, is not a pure OOP langu supports primitive data types like int and char that are not objects. In pure C even basic types are objects, and all operations are method calls.

Q5. Can we run a Java application without implementing the OOPs concept?

No. Java applications are based on Object-oriented programming models or
and hence they cannot be implemented without it. However, on the other ha
implemented without OOPs, as it also supports the C-like structural programm

Q6. What is difference between Abstraction and Encapsulation?

Abstraction is about expressing external simplicity while encapsulation is about
complexity. Eg: For a smartphone, we provide users buttons and touchscr
different features of device which is example of abstraction. While we
components like processor, memory, battery also packed in the phones body th
together to perform different tasks, thus this packing of internal components is

Q7. How is an Abstract class different from an Interface?

Though both contains only the method declaration and not the actual implem
act differently. When an interface is implemented, the subclass must define all
and provide its implementation. However when an abstract class is inherited
does not need to provide the definition of its abstract method, until and unless
using it.

**Python do not have concept of Interface

Q8. What is difference between method overloading and overriding?

Method Overloading: Involves defining multiple methods in the same class
name but different parameters (number, type, or order). (Compile-time polym
Purpose: Enables a class to have multiple methods with similar functionality but
of accepting and processing inputs.

Method Overriding: Involves providing a new implementation for a method in
is already present in its superclass. (Runtime polymorphism)
Purpose: Allows a subclass to provide a specific implementation for a method

Q9. What are different types of polymorphism?

Compile-Time Polymorphism (Static Binding):

Definition: Occurs during compile time based on the types declared or determi
time.
Mechanism: Achieved through method overloading and operator overloading.
Decision Time: Method resolution or operator binding is done at compile time.
Example: Function or operator is selected based on the method or operator si
compilation.

Runtime Polymorphism (Dynamic Binding):

Definition: Occurs at runtime and is resolved dynamically, often associated w
and method overriding.
Mechanism: Achieved through method overriding in the context of inheritance
Decision Time: The actual method to be executed is determined during runtim
type of object.
Example: The method from the derived class is selected based on the type o
runtime.

In summary, compile-time polymorphism is resolved at compile time through
while runtime polymorphism is resolved at runtime through dynamic bin
overloading is an example of compile-time polymorphism, and method overrid
the context of inheritance, is an example of runtime polymorphism.

Q10. What is coupling in OOPs?

Coupling in Object-Oriented Programming (OOP) refers to the degree of depend
different classes or modules. It indicates how much one class or module relie
There are several types of coupling:

Low Coupling (Loose Coupling): Classes are independent, and changes in on
likely to affect another.
Benefits: Promotes reusability, maintainability, and flexibility.
Example: Using interfaces or abstract classes to interact with objects, re
dependencies.

High Coupling (Tight Coupling): Classes are highly dependent on each other, a

Q11. What is Composition in OOPs?

Composition in Object-Oriented Programming (OOP) is a design principle
contains objects of other classes, creating a "has-a" relationship. It allows for bu
objects by combining simpler ones. Unlike inheritance, which creates an "is-a
composition focuses on building functionality by assembling different compone

```python
class Engine:
    def start(self):
        print("Engine started.")

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition: Car has an Engine

    def start(self):
        print("Car starting...")
        self.engine.start()

my_car = Car()
my_car.start()
```

In this example, Car has an Engine (composition). The Car class doesn't inherit fr
class; instead, it contains an instance of the Engine class. This allows for flexibilit
design. If the behavior of the engine needs to change, it can be modified
without affecting the Car class.

Composition is beneficial for achieving modular, maintainable code, and it's o
over deep class hierarchies created through inheritance. It encourages a mo
adaptable code structure, supporting the principles of encapsulation and loose

Q12. Are there any limitations of Inheritance?

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

Q14. Can we overload the constructor?

In many object-oriented programming languages, you can overload constru destructors. Constructor overloading involves defining multiple constructors each with a different set of parameters. This allows objects to be initialized in v

```python
# In Python
class MyClass:
    def __init__(self, param1=None, param2=None):
        # Default constructor
        self.param1 = param1
        self.param2 = param2

# Using different constructors
obj1 = MyClass()
obj2 = MyClass(10)
obj3 = MyClass(20, "Hello")
```

```java
// In Java
public class MyClass {
    private int param1;
    private String param2;

    // Default constructor
    public MyClass() {
        this.param1 = 0;
        this.param2 = "Default";
    }

    // Constructor with one parameter
    public MyClass(int param1) {
        this.param1 = param1;
        this.param2 = "Default";
    }

    // Constructor with two parameters
    public MyClass(int param1, String param2) {
        this.param1 = param1;
        this.param2 = param2;
    }
}
```

Q16. What are different types of constructors in C++?

Default constructor: The default constructor is the constructor which doe
argument. It has no parameters

```
class ABC {
   int x;

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

Parameterized constructor: The constructors that take some arguments
parameterized constructors.

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

Copy constructor: A copy constructor is a member function that initializes a
another object of the same class. Helps in cloning of objects.

```
class ABC{
   int x;

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

Q17. What is difference between Copy constructor and Assignment Operator(=)

Though both are used to initialize object using another object. Copy constru
separate memory to both objects. However, assignment operator does not
memory for newly created object.

Q19. What are pure virtual functions?

Pure virtual functions or abstract functions are functions that are only declar
class. This means that they do not contain any definition in the base class a
redefined in the subclass.

Q20. What is Exception handling?

Exception is an event that occurs during the execution of a program at runtime
execution to halt. Exception primarily occurs where user wishes to do some
program does not support. Exceptions can be handled in program ahead of tir
the execution from stopping. Try-catch is the most common method used
exceptions in the program.

```
try:
    # Code that might raise an exception
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2

except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")

except ValueError:
    print("Error: Please enter valid numbers.")

else:
    # Executed if no exception is raised
    print("Result:", result)

finally:
    # Always executed, whether an exception occurred or not
    print("This block is always executed.")
```

Q21. What is lazy initialization?

Lazy initialization is a design pattern where the creation of an object or the co
value is deferred until the point at which it is first needed. This can be particular

Q22. What are Singleton Classes?

A Singleton class is a class that ensures only one instance of itself is created a global point of access to that instance. This pattern is useful when you want to to a resource, manage a global state, or coordinate actions across a system.

```java
//Singleton class in java
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Private constructor to prevent instantiation.
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // Other methods and properties...
}

// Usage
public class Main {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();
        System.out.println(obj1 == obj2);  // Output: true (both references point to th
    }
}
```

```python
#Singleton Class in Python
class Singleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

# Usage
obj1 = Singleton()
obj2 = Singleton()
```

Q23. Which will be executed first: Static or Main method?

Static block will be executed even before the compiler or interpreter looks method in the program. Hence static methods execute first.

Q24. What are friend functions?

Friend functions are a concept primarily associated with C++, and they do not and Java. In C++, a friend function of a class is a function that is not a member o is granted access to its private and protected members. This allows the frie operate on the private or protected members of the class as if it were a membe Friend functions are declared using the friend keyword in the class declaration.

```cpp
class MyClass {
    private:
        int privateVar;
    public:
        MyClass(int x) : privateVar(x) {}
        friend void friendFunction(const MyClass& obj);
};

// The friend function can access private members of MyClass
void friendFunction(const MyClass& obj) {
    std::cout << "Value of privateVar: " << obj.privateVar << std::endl;
}

int main() {
    MyClass myObject(42);
    friendFunction(myObject);
    return 0;
}
```

Q25. What are anonymous functions in Java?

In Java, an anonymous class is a class without a name. It's typically used for a often as an implementation of an interface or extension of a class. Anonymo declared and instantiated at the same time, usually in the context of method a part of another expression.

```java
interface Greeting {
    void greet();
}

public class Main {
    public static void main(String[] args) {
        // Using an anonymous class to implement the Greeting interface
        Greeting anonymousGreeting = new Greeting() {
            @Override
            public void greet() {
                System.out.println("Hello from the anonymous class!");
```