**PHP WITH OOPS CONCEPT**

**Q1.What is OOP’S**

Object-Oriented Programming (OOP) is a programming style that organizes code into "objects," which are collections of data (attributes) and functions (methods) that operate on that data.

**Object-oriented programming has several advantages over procedural programming**:

* OOP is faster and easier to execute
* OOP provides a clear structure for the programs
* OOP helps to keep the PHP code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* OOP makes it possible to create full reusable applications with less code and shorter development time

**Key Points:**

* **Objects**: Instances of classes that represent real-world entities.
* **Classes**: Blueprints for creating objects.
* **Encapsulation**:

Encapsulation is defined as the wrapping up of data member and member function under a single unit. It is the mechanism that binds together code and the data it manipulates.

In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

**Use**

1. Better control of class attributes and methods
2. Class attributes can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
3. Flexible: the programmer can change one part of the code without affecting other parts
4. Increased security of data

* **Inheritance**: Allowing one class to inherit properties and methods from another, promoting code reuse.
* **Polymorphism**:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

OR, A function have different behaviour in different place.

For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

1. Compile-Time Polymorphism(Static Polymorphism) **method overloading**
2. Run-Time Polymorphism (Dynamic Polymorphism) - **method overRidding**

* **Data** **Abstraction**:

Data abstraction is one of the most essential and important features of object-oriented programming.

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

**Q2. What is Class and object?**

**Class:**

A class is an entity that determines how an object will behave and what the object will contain.

In other words, it is a blueprint or a set of instruction to build a specific type of object.

A diagram of a class

Description automatically generated

**Objects**:

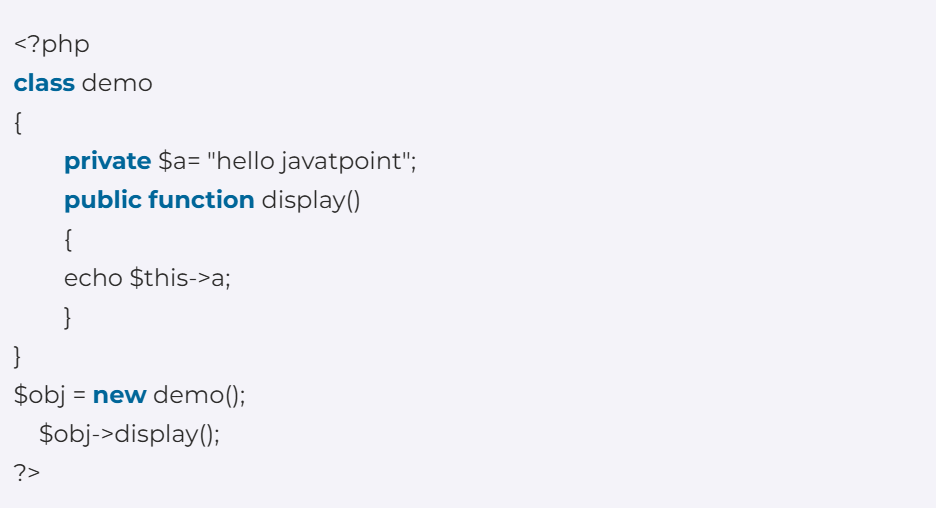
It is an instances of classes that represent real-world entities.

An object encapsulates both data (attributes) and behavior (methods) related to that data.

**Key Characteristics of an Object:**

1. **State**:
   * The state of an object is defined by its attributes (also called properties or fields). These attributes hold the data for the object. For example, a **Car** object might have attributes like **color**, **make**, and **model**.
2. **Behavior**:
   * The behavior of an object is defined by its methods (functions) that operate on the object's data. For instance, a **Car** object might have methods like **start()**, **stop()**, and **drive()**, which define what actions the car can perform.
3. **Identity**:
   * Every object has a unique identity, which distinguishes it from other objects. Even if two objects have the same state (i.e., the same attribute values), they are still considered different objects.

**Example**



**Q3. What is Constructor and Destructors?**

A **constructor** is a special member that is automatically called when an object is instantiated.

It is typically used to initialize the object's properties. Once the object is initialized, the constructor is automatically called.

We can design constructor using **"\_\_construct" or same name as class name**.

A screenshot of a computer

Description automatically generated

**Note:** The constructor is defined in the public section of the Class. Even the values to properties of the class are set by Constructors.

**Constructor types:**

1. **Default Constructor:**

It has no parameters, but the values to the default constructor can be passed dynamically.

**Pre-defined Default Constructor:** By using function \_\_construct(), you can define a constructor.  
**Note:** In the case of Pre-defined Constructor(\_\_construct) and user-defined constructor in the same class, the Pre-defined Constructor becomes Constructor while user-defined constructor becomes the normal method.

A screenshot of a computer program

Description automatically generated

A close-up of a sign

Description automatically generated

1. **Parameterized Constructor:**

The constructor of the class accepts arguments or parameters.   
The -> operator is used to set value for the variables.

In the constructor method, you can assign values to the variables during object creation.

A screen shot of a computer program

Description automatically generated

Output

A black background with white text

Description automatically generated

**Note:**

If the PHP Class has a constructor, then at the time of object creation, the constructor of the class is called.

The constructors have no Return Type, so they do not return anything not even void.

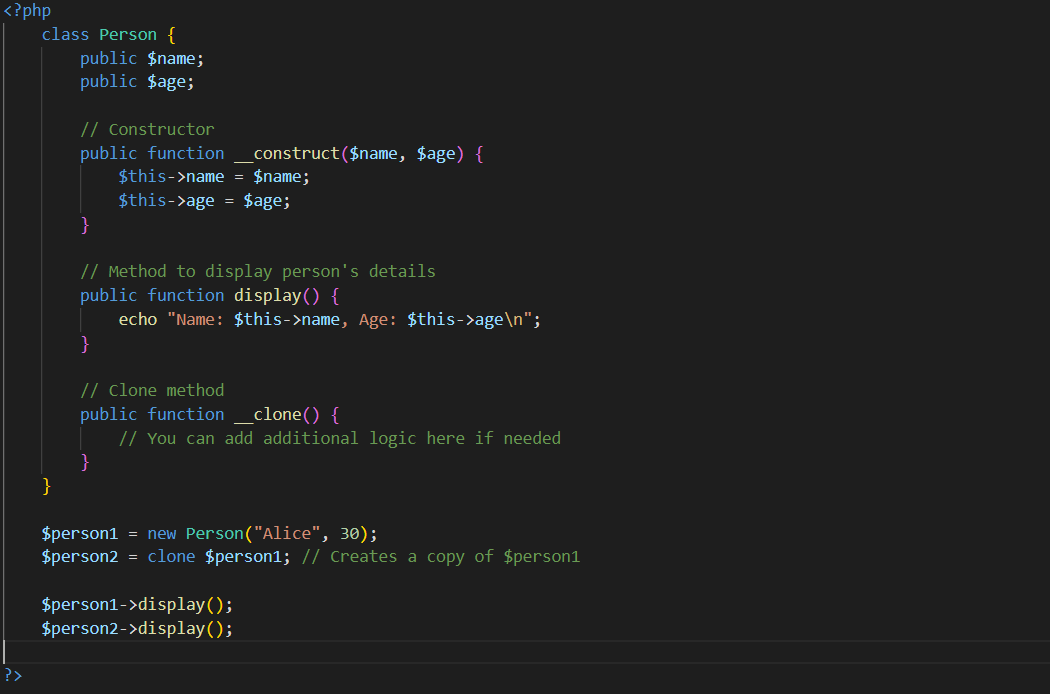
The \_\_construct() method always have the public visibility factor.

* 3. **Copy Constructor:**

It accepts the address of the other objects as a parameter.

In general we don’t have any concept of copy constructor in PHP like c++, etc.

To achieve this we use “clone” keyword to clone the object.



Output



**Advantages of using Constructors:** 

* Constructors provides the ability to pass parameters which are helpful in automatic initialization of the member variables during creation time .
* The Constructors can have as many parameters as required and they can be defined with the default arguments.
* They encourage re-usability avoiding re-initializing whenever instance of the class is created .
* You can start session in constructor method so that you don’t have to start in all the functions everytime.
* They can call class member methods and functions.
* They can call other Constructors even from Parent class.

**Destructor**

* Destructor is also a special member function which is exactly the reverse of constructor method and is called when an instance of the class is deleted from the memory.
* It is typically used to perform cleanup tasks, such as releasing resources, closing database connections, or performing any other necessary finalization before the object is removed from memory.
* The destructor is defined using the **\_\_destruct()** magic method.

**Note:**The destructor method is called when the PHP code is executed completely by its last line by using PHP **exit()** or **die() functions**.

**Note:** In the case of inheritance, and if both the child and parent Class have destructors then, the destructor of the derived class is called first, and then the destructor of the parent class.

A screen shot of a computer

Description automatically generated

**Advantages of destructors:** 

* Destructors give chance to objects to free up memory allocation , so that enough space is available for new objects or free up resources for other tasks.
* It effectively makes programs run more efficiently and are very useful as they carry out clean up tasks.
* **Comparison between \_\_constructors and \_\_destructors:**

| **Constructors** | **Destructors** |
| --- | --- |
| Accepts one or more arguments. | No arguments are passed. Its void. |
| function name is \_construct(). | function name is \_destruct() |
| It has same name as the class. | It has same name as the class with prefix ~tilda. |
| Constructor is involved automatically when the object is created. | Destructor is involved automatically when the object is destroyed. |
| Used to initialize the instance of a class. | Used to de-initialize objects already existing to free up memory for new accommodation. |
| Used to initialize data members of class. | Used to make the object perform some task before it is destroyed. |
| Constructors can be overloaded. | Destructors cannot be overloaded. |
| It is called each time a class is instantiated or object is created. | It is called automatically at the time of object deletion . |
| Allocates memory. | It deallocates memory. |
| Multiple constructors can exist in a class. | Only one Destructor can exist in a class. |
| If there is a derived class inheriting from base class and the object of the derived class is created,  the constructor of base class is created and then the constructor of the derived class. | The destructor of the derived class is called and then the destructor of base class just the reverse order of  constructor. |
| The concept of copy constructor is allowed where an object is initialized from the address of another object . | No such concept is allowed. |

**Q4.  Access Specifiers/Modifiers**

In PHP, access specifiers (also known as access modifiers) are keywords that determine the visibility and accessibility of class properties and methods.

They control how and where these properties and methods can be accessed from other parts of the code

There are three primary access specifiers **public**, **private**, and **protected**.

1. **Public**: The property or method can be accessed from anywhere, both inside and outside the class.
2. **Protected**: The property or method can be accessed only within the class itself and by inheriting classes (subclasses). It cannot be accessed from outside the class hierarchy.
3. **Private**: The property or method can be accessed only within the class that defines it. It cannot be accessed from outside the class or by subclasses.

**Example of Access Specifiers**



A screenshot of a computer program

Description automatically generated

Output

A black rectangle with white dots

Description automatically generated

**Q5. What is Inheritance in PHP?**

**Inheritance** is a fundamental concept in object-oriented programming (OOP) that allows a class (known as a **child class** or **subclass**) to inherit properties and methods from another class (known as a **parent class** or **superclass**).

This mechanism promotes code reusability and establishes a natural hierarchy between classes.

**Key Concepts of Inheritance**

1. **Parent Class**: The class from which properties and methods are inherited.
2. **Child Class**: The class that inherits from the parent class.
3. **Method Overriding**: A child class can provide a specific implementation of a method that is already defined in its parent class.
4. **Access to Parent Methods**: Child classes can access public and protected methods of the parent class, but not private methods.

A screen shot of a computer

Description automatically generated

Output



**Explanation**

1. **Parent Class**: Animal is the base class with a method speak().
2. **Child Classes**: Dog and Cat extend Animal and override the speak() method to provide specific sounds.
3. **Method Overriding**: When speak() is called on a Dog or Cat object, the overridden method in the respective class is executed.

**Benefits of Inheritance**

* **Code Reusability**: Common functionality can be defined in a parent class and reused in child classes, reducing code duplication.
* **Logical Structure**: Inheritance helps create a clear hierarchical relationship between classes, making the codebase easier to understand and maintain.
* **Polymorphism**: Inheritance supports polymorphism, allowing different classes to be treated as instances of the same parent class, making code more flexible and extensible.

**Types of Inheritance**

In PHP, inheritance can be categorized into several types based on how classes relate to one another. Here are the main types of inheritance:

* **Single Inheritance**
* **Multilevel Inheritance**
* **Hierarchical Inheritance**
* **Multiple Inheritance**

**A diagram of a class

Description automatically generated**

**1. Single Inheritance**

A class can inherit from only one parent class. This is the most straightforward form of inheritance.

****

**2. Multilevel Inheritance**

In multilevel inheritance, a class inherits from a parent class, and then another class inherits from that child class. This creates a chain of inheritance.

Means Class C inherit class B and class B Inherit class C, class A is parent and B, C is child

A screen shot of a computer program

Description automatically generated

**3. Hierarchical Inheritance**

In hierarchical inheritance, multiple child classes inherit from a single parent class. Each child class can have its own unique properties and methods.

****

**4. Multiple Inheritance (via Interfaces)**

While PHP does not support multiple inheritance of classes, it does allow a class to implement multiple interfaces. This allows for a form of multiple inheritance by defining contracts that a class must adhere to.

**A screen shot of a computer program

Description automatically generated**

**Why PHP not support Multiple Inheritence**

**Ambiguity in Method Resolution:**

* One of the main challenges with multiple inheritance is the diamond problem, where a class inherits from two (or more) classes that share a common ancestor. If both parent classes define the same method, it can create ambiguity about which method to call.

**Q6.Method Overriding**

When a child class defines a method with the same name, same parameters, and same return type as a method in its parent class, it is said to override that method.

This allows the child class to provide a specific implementation for that method.

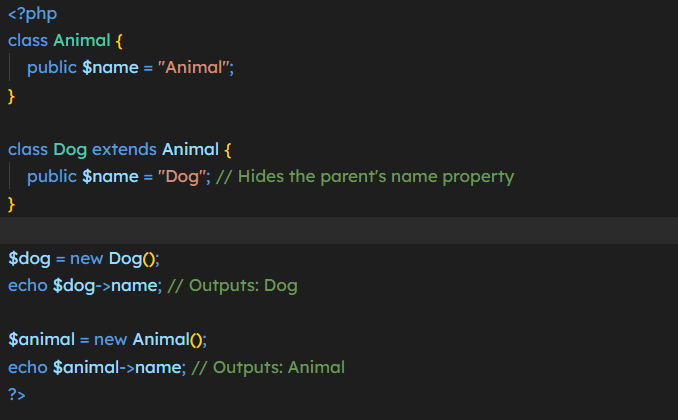
A screen shot of a computer program

Description automatically generated

**Property Overriding**

In PHP, properties cannot be overridden in the same way methods can.

If a child class defines a property with the same name as a property in the parent class, the child class's property will hide the parent class's property, but it does not override it in the traditional sense.



**Key Points About Property Overriding**

1. **Hiding**: When a child class declares a property with the same name as a parent class property, it hides the parent property, but it does not override it. The parent property is still accessible via an instance of the parent class.
2. **Different Values**: Each class can have its own value for the property, and the child class's property will take precedence when accessed through an instance of the child class.
3. **Access to Parent Method**: If you want to use the parent class's version of a method in the child class, you can do so using the parent::methodName()

**Difference Between Method Overloading and Method Overriding**

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is a compile-time polymorphism. | Method overriding is a run-time polymorphism. |
| Method overloading helps to increase the readability of the program. | Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass. |
| It occurs within the class. | It is performed in two classes with inheritance relationships. |
| Method overloading may or may not require inheritance. | Method overriding always needs inheritance. |
| In method overloading, methods must have the same name and different signatures. | In method overriding, methods must have the same name and same signature. |
| In method overloading, the return type can or can not be the same, but we just have to change the parameter. | In method overriding, the return type must be the same or co-variant. |
| Static binding is being used for overloaded methods. | Dynamic binding is being used for overriding methods. |
| Private and final methods can be overloaded. | Private and final methods can’t be overridden. |
| The argument list should be different while doing method overloading. | The argument list should be the same in method overriding. |

**6.1 Final Methods**

In PHP, the final keyword is used to prevent certain actions from being performed. It can be applied to methods, classes, and class properties with the following purposes:

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

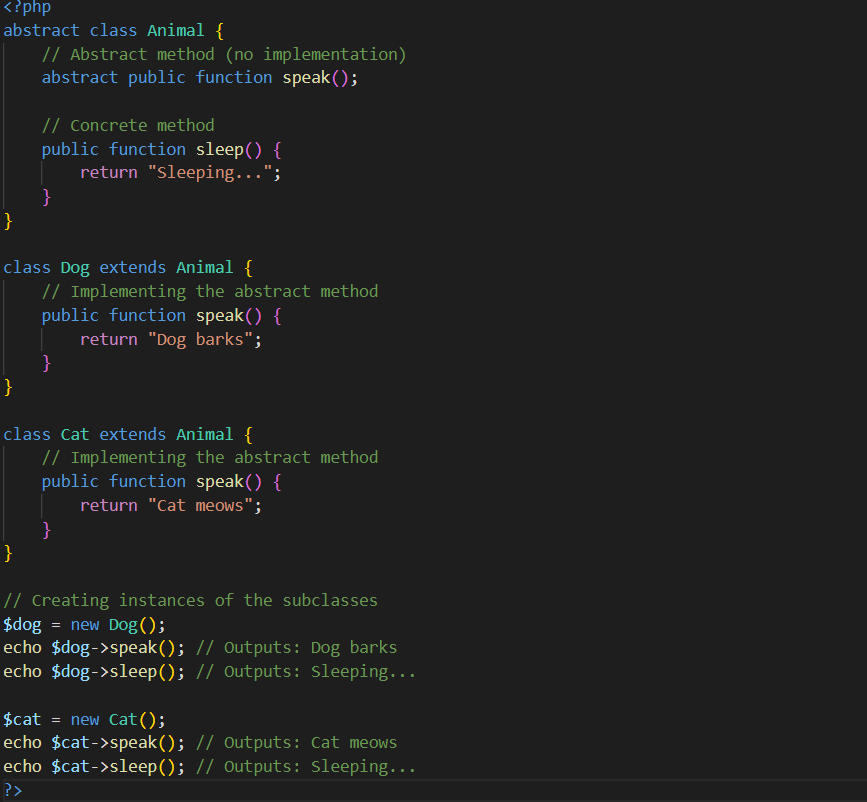
**Q7. Abstract Classes in PHP**

Abstract class is a class that cannot be instantiated on its own .Instead, they serve as a blueprint for other classes to extend and implement their abstract methods.

We can not create an object of this class.

**Points to remember**

1. To create an abstract class always use “abstract” Keyword.
2. Always declare an abstract method.
3. Extend the abstract class and define the abstract method in subclass.



**When to Use Abstract Classes**

* **Common Interface**: Use abstract classes when you have a common interface for a group of related classes, but you don't want to instantiate the base class itself.
* **Shared Functionality**: If you want to provide some shared functionality (concrete methods) that can be used by all subclasses, abstract classes are a good choice.
* **Enforcing Method Implementation**: If you want to ensure that certain methods are implemented in all subclasses, you can declare them as abstract methods.

**Q8. What is Interfaces**

In PHP, an **interface** is a contract that defines a set of methods that a class must implement, but it does not provide any implementations for those methods.

Interfaces are a way to achieve polymorphism and define a common behavior that multiple classes can share, regardless of their position in the class hierarchy.

**Key Features of Interfaces**

1. **Method Declaration**: An interface can declare methods, but it cannot implement them. All methods in an interface are abstract by default.
2. **No Properties**: Interfaces cannot contain properties or constants (though they can declare constants).
3. **Multiple Inheritance**: A class can implement multiple interfaces, which allows for more flexibility than class inheritance. A computer screen shot of code

   Description automatically generated
4. **Access Modifiers**: All methods in an interface must be declared as public.

**Note** : To define an interface, use the interface keyword

A screen shot of a computer program

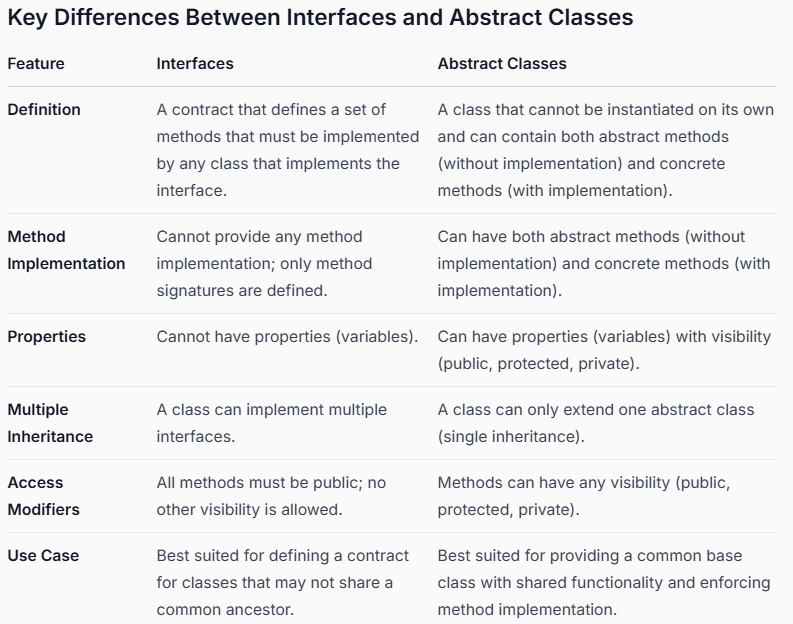
Description automatically generated

**Explanation of the Example**

1. **Interface (Animal)**: The **Animal** interface defines two methods: **speak()** and **eat()**. These methods do not have any implementations; they only specify the method signatures.
2. **Implementing Classes (Dog and Cat)**: Both the **Dog** and **Cat** classes implement the **Animal** interface. They provide concrete implementations for the **speak()** and **eat()** methods.
3. **Creating Instances**: You can create instances of the **Dog** and **Cat** classes and call their methods, which will execute the specific implementations provided by each class.

**When to Use Interfaces**

* **Defining Contracts**: Use interfaces when you want to define a contract that multiple classes must adhere to. This is useful for ensuring that certain methods are implemented across different classes.
* **Multiple Inheritance**: When you need a class to inherit behavior from multiple sources, interfaces allow you to achieve this since a class can implement multiple interfaces.
* **Decoupling Code**: Interfaces help decouple code by allowing you to define behaviors that can be implemented in various ways, making your code more flexible and easier to maintain.



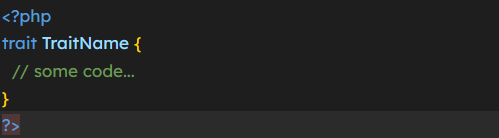
**Q9. What are Traits?**

Traits are used to declare methods that can be used in multiple classes without Inheritance.

Traits can have methods and abstract methods that can be used in multiple classes, and the methods can have any access modifier (public, private, or protected).

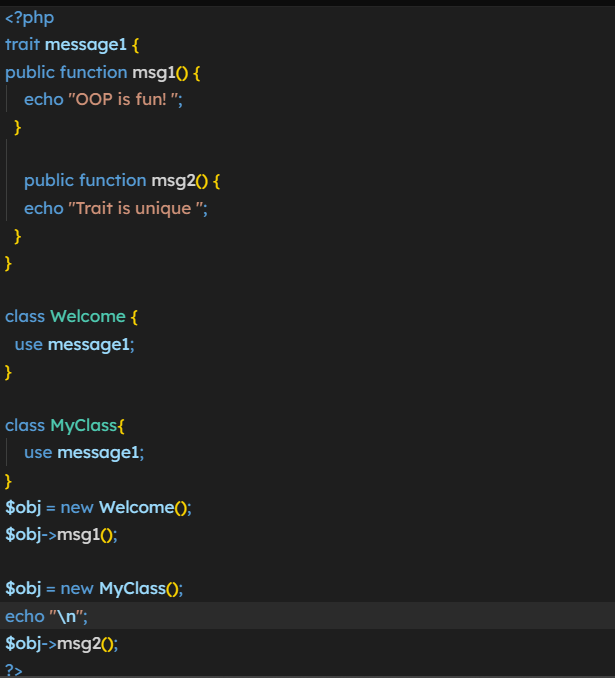
Traits help in overcoming limitations of single inheritance in PHP, enabling a class to use methods from multiple sources.

We can create trait using “trait” keyword same as class.



**Key Features of Traits:**

1. **Reusability**: Traits allow you to define methods that can be reused in multiple classes without the need for inheritance.
2. **No State**: Traits are typically stateless. They can contain methods and properties, but they are not meant to represent a complete object.
3. **Method Conflict Resolution**: If two traits define a method with the same name, you can use the insteadof operator to resolve the conflict when using them in a class.
4. **Flexible Composition**: You can compose classes from multiple traits, making it easy to add functionality without a rigid class hierarchy.



**Overriding Trait Methods**

When a class uses a trait, it can override any methods defined in the trait. If a class defines a method with the same name as a method in the trait, the class's method will take precedence.



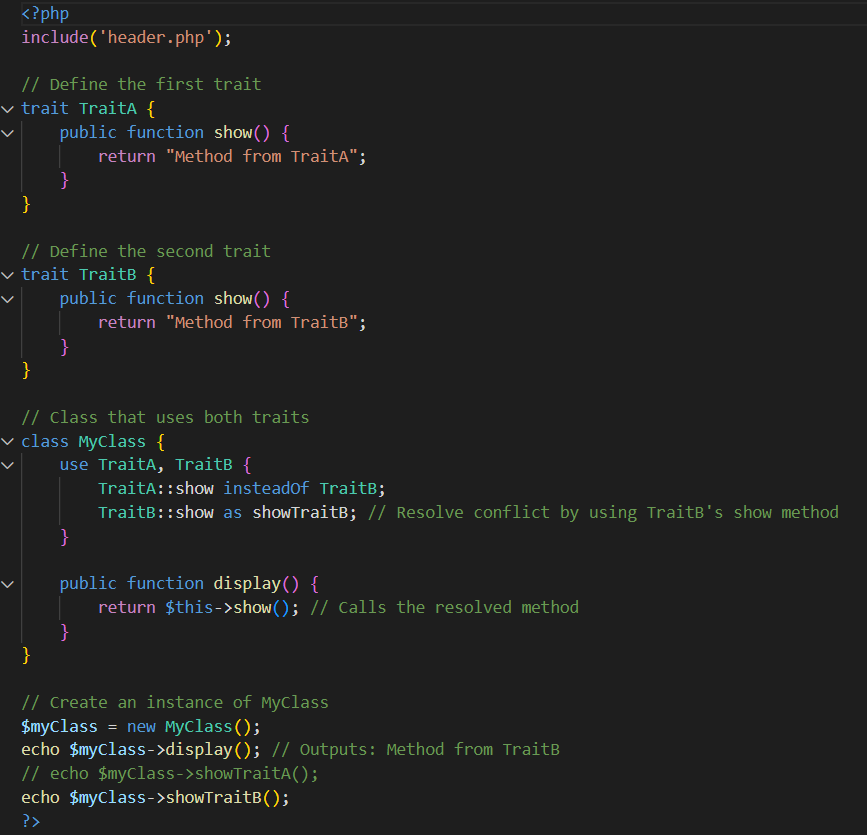
**Explanation**

1. **Trait Definition**: The **Greeting** trait defines a method **sayHello()** that returns a greeting message.
2. **Base Class**: The **BaseClass** also defines a method **sayHello()** that returns a different message.
3. **Using the Trait**: The **MyClass** class extends **BaseClass** and uses the **Greeting** trait. It also defines its own version of the **sayHello()** method, which overrides both the trait's method and the base class's method.
4. **Method Resolution**: When the **sayHello()** method is called on an instance of **MyClass**, the output is "Hello from MyClass!", demonstrating that the method in the class takes precedence over the methods in both the trait and the base class.

**Example of Multiple Traits with the Same Method Name**

When you have multiple traits that define methods with the same name and you want to use them in a single class, you will encounter a conflict.

PHP provides a way to resolve this conflict using method aliasing, allowing you to give different names to the methods from the traits so that you can call them without ambiguity.



**Explanation**

A screenshot of a computer

Description automatically generated

**Q10. What is Static Methods ?**

Static methods are the methods that can be called directly - without creating an instance of the class first.

Static methods are declared with the static keyword:

Instead of this we use self keyword to and use :: to call static method.

**A computer screen shot of text

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**What is the difference between self:: and parent:: in PHP?**

** self**::: Refers to the current class. It is used to access static properties, static methods, and constants within the class where it is written.

** parent:::** Refers to the parent class. It is used to access parent class methods or parent class properties, even if the method is overridden in the child class**.**

**Q11. Late Static Binding**

In PHP, static methods are traditionally bound to the class in which they are defined, rather than the class from which they are called.

Late Static Binding provides a way to reference the class that is actually being executed, which is particularly useful in the context of inheritance and static methods.

IMP- LSB allows the static method to resolve to the **class that was actually called** at runtime, which could be a subclass.

The key to LSB is the static keyword, which is used in conjunction with the self or parent keyword.

A computer screen shot of code

Description automatically generated

Explanation

In this example there are two classes

1st base and 2nd derived

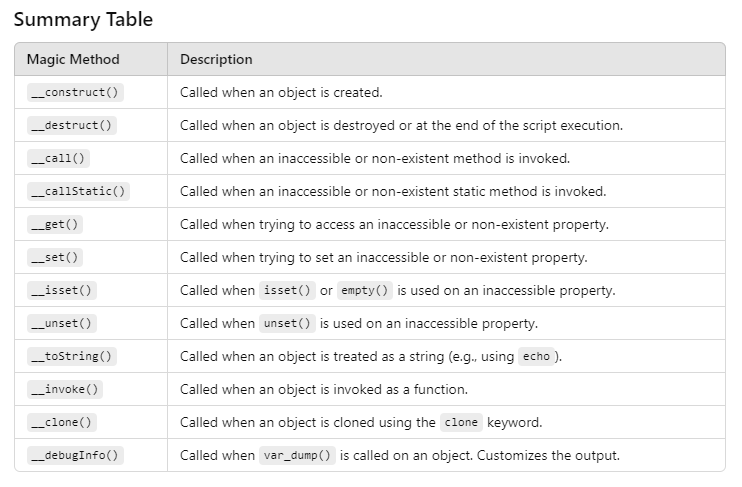
So when we can create a object of derived class and called the method show of base class then it will print the $name of base class.

But by using static keyword and :: operator it will print the $name of derived class which is called at run time.

**Q12. Magic Methods in PHP**

Magic methods are special methods in PHP that allow you to perform certain actions when specific events occur, such as object creation, property access, method calls, and destruction. These methods are called "magic" because they are invoked automatically by the PHP runtime when certain operations are performed.

Magic methods in PHP start with double underscores (\_\_), and they help make objects more flexible and customizable.



**Design Pattern IN OOP’S**

**Q13. What is Dependency Injection ?**

Dependency Injection (DI) is a design pattern used in object-oriented programming that allows a class to receive its dependencies from an external source rather than creating them internally.

It is used to achieve **Inversion of Control (IoC)** between classes and their dependencies.

**Advantages of Dependency Injection:**

* **Loose Coupling**: Classes are less tightly coupled to their dependencies. This makes code easier to maintain and extend.
* **Testability**: It’s easier to mock or substitute dependencies in unit tests.
* **Flexibility**: The dependent objects can be easily replaced or modified without changing the dependent class.

**Key Concepts of Dependency Injection**

1. **Dependencies**: These are the objects that a class needs to function. For example, if a class **User Service** requires a **User Repository** to fetch user data, then **User Repository** is a dependency of **User Service**.
2. **Injection**: This is the process of providing the dependencies to a class. There are several ways to inject dependencies:

**How Dependency Injection Works:**

* A class that depends on another class does not create the dependent class instance itself. Instead, it accepts the dependency from outside (through the constructor, method, or property).

**DI can be achieved in various ways:**

* 1. **Constructor Injection**: Passing dependencies via the class constructor.
  2. **Setter Injection (or Method Injection)**: Passing dependencies via setter methods.
  3. **Property Injection**: Injecting dependencies directly into the class properties

**Example :**

**Constructor Injection**A screenshot of a computer program

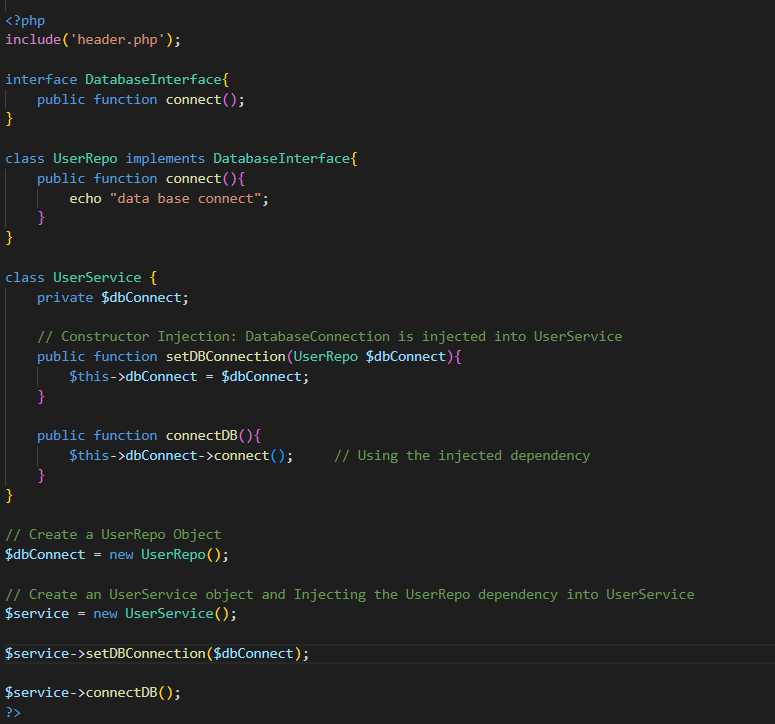
Description automatically generated

**$service->connectDB();**

Example : Setter Method

In this example we us DI with the help of setter method so the difference is that instead of creating Contructor we create a setter method and put the DI in parameter of setter method.

When We create an object of class in which setter method is define that is UserService after creating object call setter method and inject DI.



**Q14. Singleton Design Pattern**

The Singleton pattern is a design pattern in OOP’S that ensures a class has only one instance and provides a global point of access to that instance.

This pattern is particularly useful when exactly one object is needed to coordinate actions across the system, such as in cases where you want to manage shared resources like configuration settings, logging, or database connections.

**Key Characteristics of the Singleton Pattern**

1. **Single Instance**: The Singleton pattern ensures that a class has only one instance throughout the application.
2. **Global Access**: It provides a global point of access to that instance, allowing other parts of the application to use it without needing to create new instances.
3. **Lazy Initialization**: The instance is usually created only when it is needed, which can help save resources.

**Rules to create Singleton Design Pattern**

**1.Private static variable**

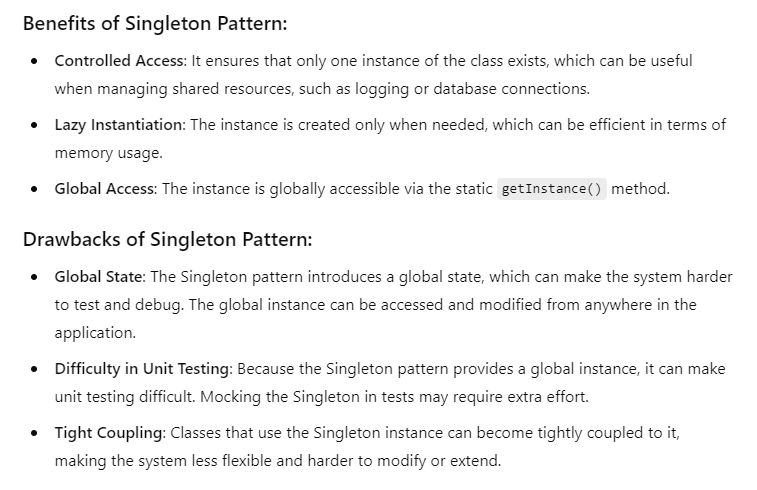
**2.Private constructor**

**3.Public static method which return class singleton instance**

**Example**

**A computer screen shot of text

Description automatically generated**

****

**Q15. Factory Design pattern**

**In**[**object-oriented programming**](https://en.wikipedia.org/wiki/Object-oriented_programming)**, the factory method pattern is a**[**design pattern**](https://en.wikipedia.org/wiki/Software_design_pattern)**that uses factory methods to deal with the problem of**[**creating objects**](https://en.wikipedia.org/wiki/Object_creation)**without having to specify their exact**[**classes**](https://en.wikipedia.org/wiki/Class_(computer_programming))**.**

**OR**

The Factory Method Pattern is a design pattern in object-oriented programming that allows you to create objects without specifying the exact class of the object that will be created.

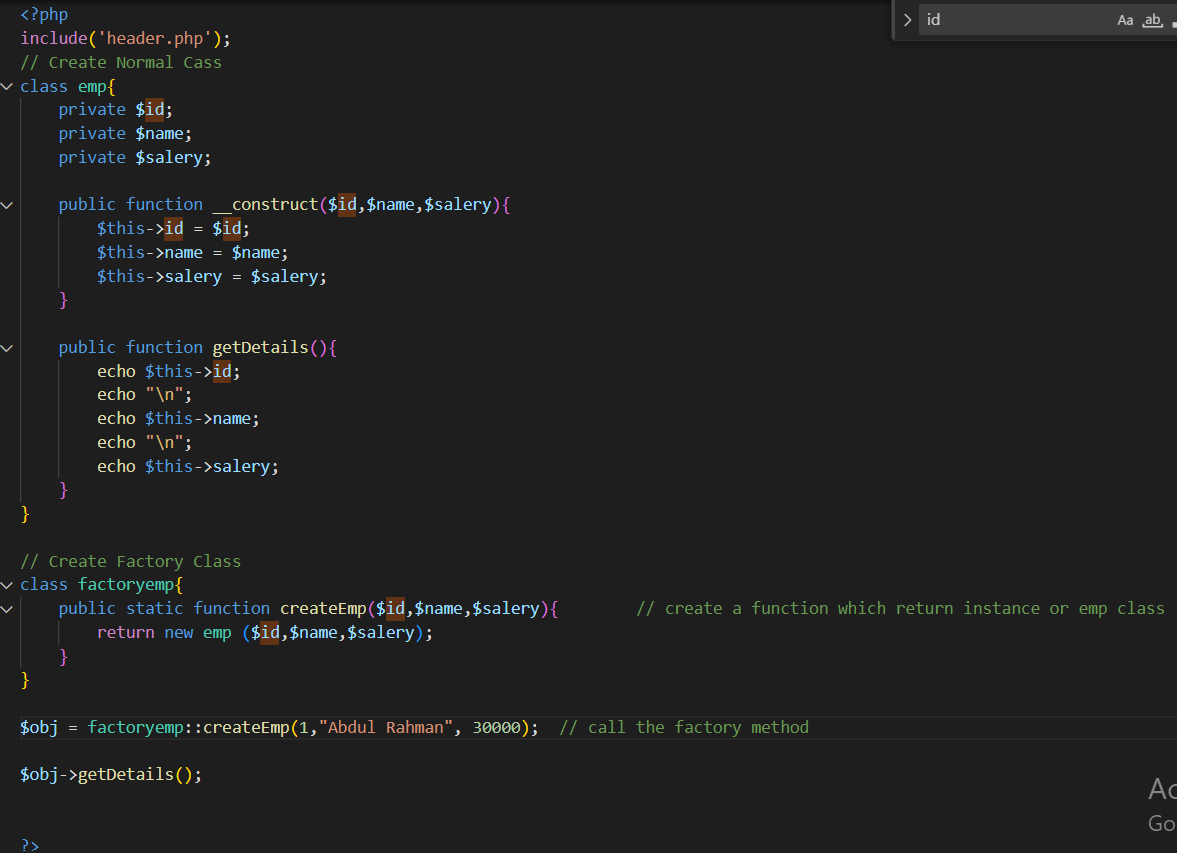
Instead of calling a constructor directly, you use a factory method that returns an instance of a class.

This pattern helps to promote loose coupling and makes it easier to add new types of objects in the future.

**Key Points:**

* **Encapsulation**: The creation logic is encapsulated in a method, making it easier to manage.
* **Flexibility**: You can easily change or extend the types of objects created without modifying existing code.
* **Loose Coupling**: The code that uses the objects is decoupled from the specific classes of the objects.

Example 1



Example 2 – Using Interface

A screen shot of a computer program

Description automatically generated

**SOLID Principle of OOPS**

The SOLID principles are a set of five principles that are designed to make object-oriented programming (OO) more understandable, flexible, and maintainable.

1. **Single Responsibility Principle (SRP)**

**Definition**: A class should have only one reason to change, meaning that it should only have one job or responsibility.

* **Explanation**: If a class is responsible for multiple tasks, it is likely that changes in one task could affect other tasks, leading to complex code. Keeping classes focused on one responsibility makes the system easier to maintain and extend.

A screen shot of a computer program

Description automatically generated

**Explanation**

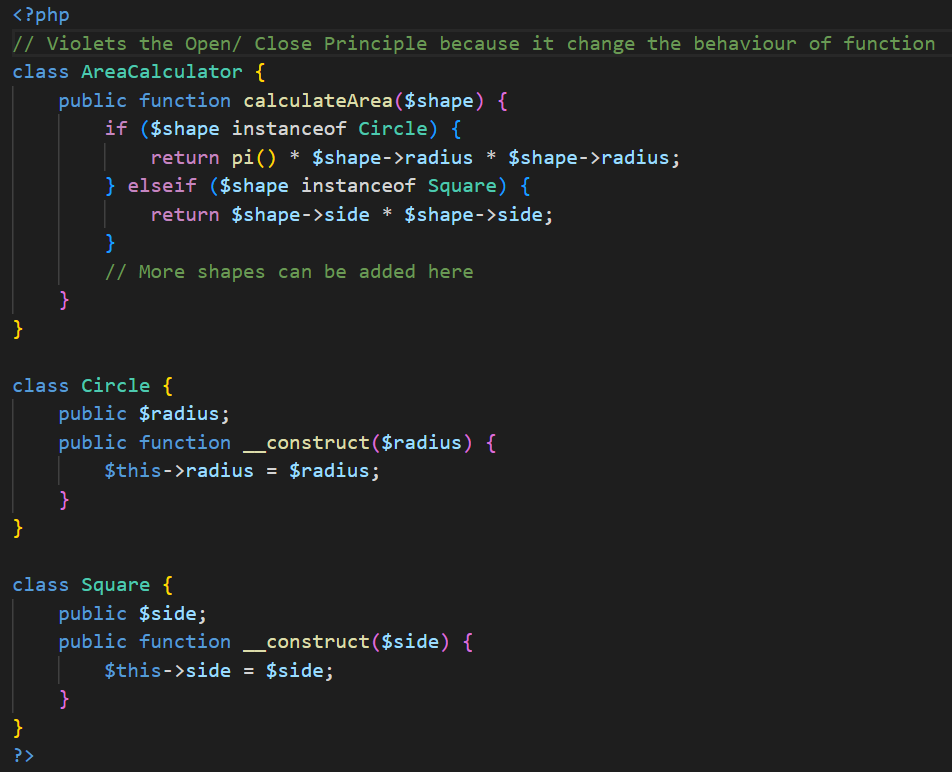
In this example, the User class has multiple responsibilities (saving data and logging activity), which is violating the SRP.

The solution is to separate these concerns into two classes: User and Logger.

**2. Open/Closed Principle (OCP)**

**Definition:** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

* **Explanation:** You should be able to add new functionality to a class without changing its existing code.
* This is usually achieved through inheritance or interfaces.

****

**In the above example,** if you want to add a new shape (e.g., Rectangle), you would need to modify the AreaCalculator class, which violates the Open/Closed Principle.

**Example of O/C Principle**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**Explanation of the Refactored Code**

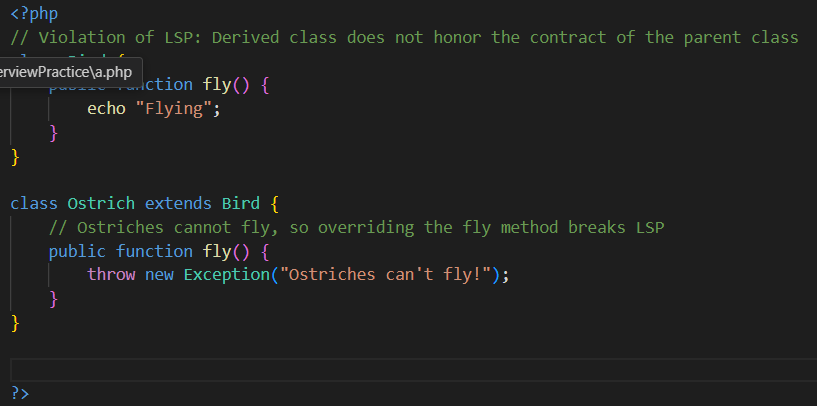
1. Interface: We define a Shape interface with an area() method. Each shape class (e.g., Circle, Square, Rectangle) implements this interface.
2. Area Calculation: The AreaCalculator class now only needs to call the area() method on the Shape interface. It does not need to know the details of how each shape calculates its area.
3. Adding New Shapes: If you want to add a new shape (like Rectangle), you simply create a new class that implements the Shape interface without modifying the AreaCalculator class.

**3. Liskov Subsitution Principle**

The Liskov Substitution Principle (LSP)  states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

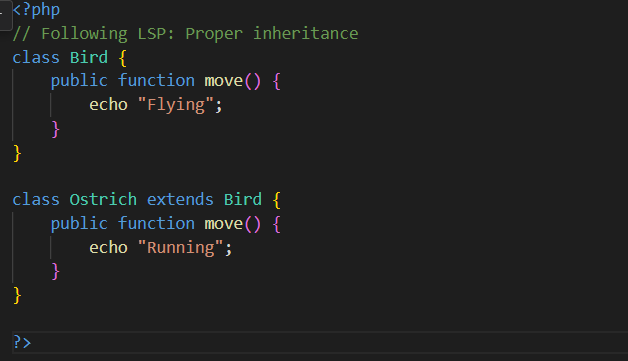
In simpler terms, if class S is a subclass of class T, then objects of type T should be replaceable with objects of type S without altering any of the desirable properties of the program (correctness, task performed, etc.).

**Voilation of LSP**



In the first example, Ostrich breaks the **Liskov Substitution Principle** because it changes the expected behavior of the fly() method. In the second example, both Bird and Ostrich implement a move() method, where Ostrich substitutes the behavior by running instead of flying.

Correct one which Follow LSP Principle

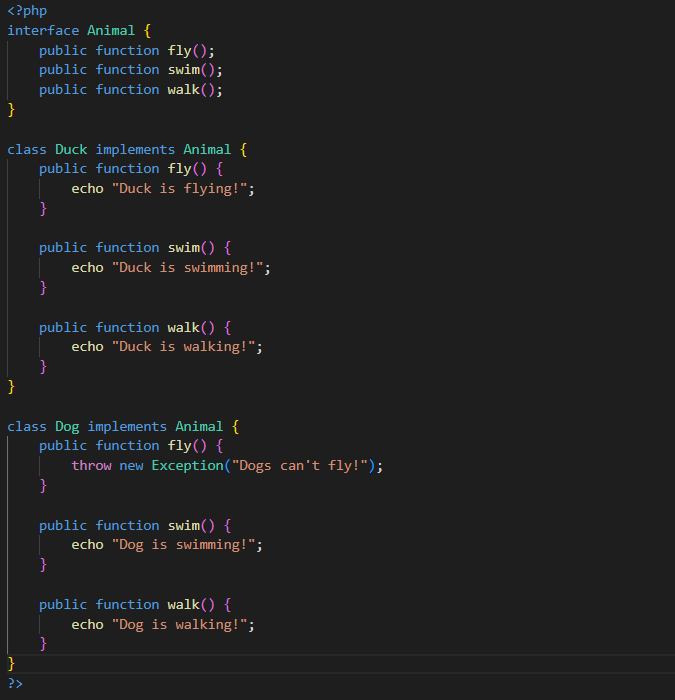


1. **Interface Segregation Principle (ISP)**

**Definition**: Clients should not be forced to depend on interfaces they do not use.

* **Explanation**: Instead of having a large, general-purpose interface, you should create smaller, more specific interfaces so that clients are only required to implement the methods they need.

Example of Violation ISP



Example : Follow ISP

A screen shot of a computer program

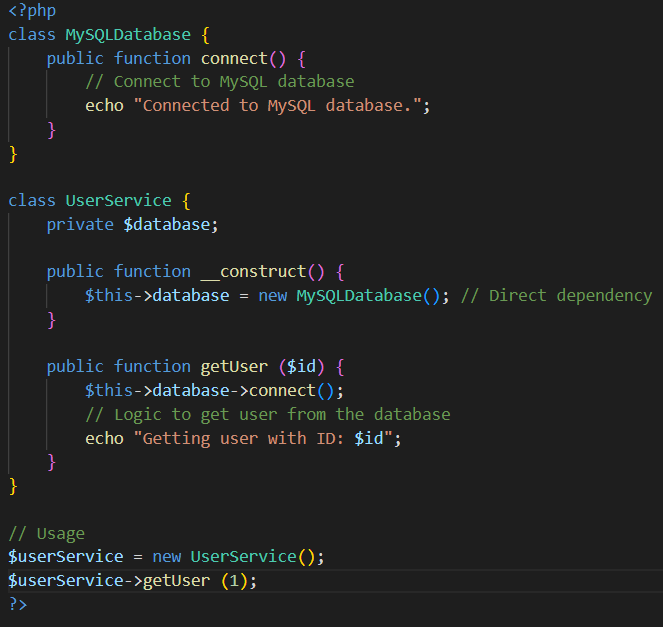
Description automatically generated

**5. Dependency Inversion Principle (DIP)**

1. High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces or abstract classes).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions

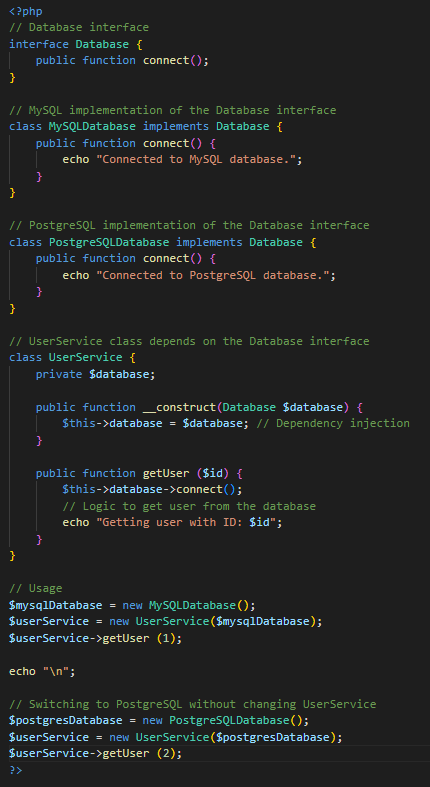
**Example**

**Without DIP Violation**

****

In this example, the User Service class is tightly coupled to the MySQLDatabase class. If you want to change the database implementation (e.g., to a PostgreSQL database), you would need to modify the User Service class.

**Correct Implementation of DIP**



**What is Object Relational Mapping (ORM)**

ORM (Object-Relational Mapping) is a technique that allows developers to interact with a relational database using an object-oriented paradigm.

With ORM, developers can map database tables to objects in their code, making it easier to interact with the database without writing raw SQL queries.

Only For Zend

In the context of **Zend Framework** (now known as **Laminas**), ORM isn't directly part of the framework itself, but ORM is generally associated with **Doctrine ORM**, which is a powerful ORM library commonly used with Zend Framework (and now Laminas) projects.