Model-4

1. What Is Object Oriented Programming?

-> Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects rather than functions or logic. These objects represent real-world entities or concepts and contain both data (fields or attributes) and methods (functions or operations) that act on the data.

-> Concepts in OOP:

Class:

Object:

Encapsulation:

Abstraction:

Inheritance:

Polymorphism:

2. What Are Properties Of Object Oriented Systems?

-> Object-Oriented Systems (OOS) are based on the principles of Object-Oriented Programming (OOP), which emphasizes using objects—instances of classes—to model real-world systems. The key properties of object-oriented systems include:

1. Encapsulation

Definition: Encapsulation refers to bundling data (attributes) and methods (functions) that operate on the data into a single unit called an object. It restricts direct access to an object’s internal state and forces interaction with the object through its methods.

Benefits: Enhances modularity, hides the complexity of implementation, and protects object integrity by preventing unauthorized access to data.

2. Abstraction

Definition: Abstraction involves hiding the complex reality while exposing only the essential parts. It allows focusing on what an object does, instead of how it does it.

Benefits: Reduces complexity by allowing a simplified interface to interact with objects, making systems easier to understand and maintain.

3. Inheritance

Definition: Inheritance allows one class (the child or subclass) to inherit attributes and methods from another class (the parent or superclass). This supports code reuse and the creation of hierarchical relationships.

Benefits: Promotes reusability and can reduce redundancy by allowing common functionalities to be shared across multiple classes.

4. Polymorphism

Definition: Polymorphism allows objects of different classes to be treated as objects of a common superclass, typically through shared methods. It enables the same method to behave differently based on the object calling it.

Types:

Compile-time polymorphism (Method Overloading)

Run-time polymorphism (Method Overriding)

Benefits: Promotes flexibility and scalability, enabling code to be more dynamic and easier to extend.

5. Modularity

Definition: Modularity refers to dividing a program into separate, independent units or modules, each with a specific functionality. Objects represent individual modules, interacting with each other to achieve the system's overall behavior.

Benefits: Improves code organization, makes debugging easier, and allows different teams to work on different modules in parallel.

6. Dynamic Binding

Definition: Dynamic binding (also known as late binding) occurs when the method to be invoked is determined at runtime based on the object type. This enables polymorphism and method overriding in object-oriented systems.

Benefits: Increases flexibility by allowing programs to decide which method to call at runtime, facilitating more dynamic behavior in systems.

7. Message Passing

Definition: Objects communicate with each other through message passing, where one object sends data (messages) to another object to invoke methods or change the state.

Benefits: Decouples object interactions, making it easier to modify or extend the system without affecting unrelated components.

These properties collectively make object-oriented systems more maintainable, scalable, and flexible compared to procedural programming approaches.

3. What Is Difference Between Class And Interface?

| **Class** | **Interface** |
| --- | --- |
| The keyword used to create a class is “class” | The keyword used to create an interface is “interface” |
| A class can be instantiated i.e., objects of a class can be created. | An Interface cannot be instantiated i.e. objects cannot be created. |
| Classes do not support multiple inheritance. | The interface supports multiple [inheritance.](https://www.geeksforgeeks.org/inheritance-in-java/) |
| It can be inherited from another class. | It cannot inherit a class. |
| It can be inherited by another class using the keyword ‘extends’. | It can be inherited by a class by using the keyword ‘implements’ and it can be inherited by an interface using the keyword ‘extends’. |
| It can contain constructors. | It cannot contain constructors. |
| It cannot contain abstract methods. | It contains abstract methods only. |
| Variables and methods in a class can be declared using any access specifier(public, private, default, protected). | All variables and methods in an interface are declared as public. |
| Variables in a class can be static, final, or neither. | All variables are static and final. |

4. What Is Overloading?

-> Overloading in Object Oriented Programming is a fundamental concept in object-oriented programming, allowing a class to define multiple methods with the same name but varying parameters. This technique enables developers to perform different operations based on the type and number of arguments provided, streamlining code organization and enhancing its flexibility.

5. What Is T\_PAAMAYIM\_NEKUDOTAYIM (Scope Resolution Operator (::) with

Example

-> T\_PAAMAYIM\_NEKUDOTAYIM is the name for the scope resolution operator

(::) in PHP. The name comes from Hebrew, meaning "double colon," and is

used to access static, constant, and overridden properties or methods

 of a class without creating an instance of that class.

 1. Accessing Static Members:

 <?php

class MyClass {

    public static $staticProperty = "I am a static property";

    public static function staticMethod() {

        return "I am a static method";

    }

}

// Accessing static property and method using the scope resolution operator

echo MyClass::$staticProperty; // Output: I am a static property

echo MyClass::staticMethod();  // Output: I am a static method

?>

2. Accessing Class Constants:

<?php

class MyClass {

    const MY\_CONSTANT = "I am a constant";

    public function showConstant() {

        return self::MY\_CONSTANT; // Accessing constant within the class

    }

}

// Accessing constant outside the class using the scope resolution operator

echo MyClass::MY\_CONSTANT;  // Output: I am a constant

?>

6. What are the differences between abstract classes and interfaces?

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1. Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods.  Since Java 8, it can have **default**  **and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the**  **implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare  interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java  interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using  keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by  default. |

7. Define Constructor and Destructor?

* contructor

A constructor is a member function of a class that initializes the object and allocates the memory. A constructor has the same name as that of its class, thus it can be easily identified. It is always declared and defined in the public section of a class. A constructor does not have any return type. Therefore, it does not return anything, but it is not even void.

Snytex:

\_\_construct([mixed](https://www.php.net/manual/en/language.types.mixed.php) ...$values = ""): [void](https://www.php.net/manual/en/language.types.void.php)

-> Destructor

A destructor is a member function of a class that deallocates the memory allocated to an object. A destructor is also declared and defined with the same name as that of the class. A single class has only a single destructor.

Snytex:

\_\_destruct(): [void](https://www.php.net/manual/en/language.types.void.php)

8. Are Parent Constructor Called Implicitly When Create An ObjectOf Class?

**parent constructors are not called implicitly** when you create an object of a child class. If the parent class has a constructor and you want it to be executed when an object of the child class is created, you must explicitly call the parent constructor using parent::\_\_construct() within the child class constructor.

Ex

<?php

class ParentClass {

public function \_\_construct()

{

echo "Parent constructor called\n";

}

}

class ChildClass extends ParentClass {

public function \_\_construct() {

echo "Child constructor called\n";

// Parent constructor is not called implicitly

}

}

$child = new ChildClass();

// Output: Child constructor called

?>

9. What Happen, If Constructor Is Defined As Private Or Protected?

a **constructor** is defined as **private** or **protected** in PHP, it affects how and where objects of that class can be instantiated. This provides control over object creation and is often used in design patterns.

1. **Private Constructor**:

* A **private constructor** prevents direct instantiation of a class from outside the class. It can only be called from within the class itself.
* This is commonly used in the **Singleton design pattern**, where only a single instance of a class is allowed.

ex

<?php

class MyClass {

private function \_\_construct() {

echo "Private constructor called\n";

}

public static function createInstance() {

return new MyClass();

// Constructor can be called from within the class

}

}

// $obj = new MyClass();

// Error: Cannot instantiate because constructor is private

$obj = MyClass::createInstance();

// Works: Creates an instance via a static method

?>

**2. Protected Constructor:**

* A **protected constructor** allows instantiation of a class within the class itself or from its **subclasses**, but not from outside.
* It provides a way to allow inheritance but still control direct instantiation.

Ex

<?php

class ParentClass {

protected function \_\_construct() {

echo "Protected constructor called\n";

}

public static function createInstance() {

return new ParentClass(); // Can create instance within the class

}

}

class ChildClass extends ParentClass {

public function \_\_construct() {

parent::\_\_construct(); // Calling the protected constructor from subclass

echo "ChildClass constructor called\n";

}

}

// $obj = new ParentClass(); // Error: Cannot instantiate because constructor is protected

$child = new ChildClass(); // Works: Child class can call protected constructor

?>

10. What are PHP Magic Methods/Functions? List them Write program for Static Keyword in PHP?

**Magic Methods** are special methods that are automatically called in certain situations. They all start with two underscores (\_\_) and are predefined by PHP. Magic methods provide a way to hook into the object-oriented behavior of PHP, allowing for more dynamic behavior in classes.

Used Magic Methods:

**\_\_construct()**: Invoked when a new object is instantiated.

**\_\_destruct()**: Invoked when an object is destroyed or goes out of scope.

**\_\_get($name)**: Invoked when accessing inaccessible or non-existent properties.

**\_\_set($name, $value)**: Invoked when setting inaccessible or non-existent properties.

**\_\_isset($name)**: Invoked when calling isset() or empty() on inaccessible properties.

**\_\_unset($name)**: Invoked when calling unset() on inaccessible properties.

**\_\_sleep()**: Invoked when serializing an object (e.g., using serialize()).

**\_\_wakeup()**: Invoked when unserializing an object (e.g., using unserialize()).

**\_\_toString()**: Invoked when an object is treated like a string (e.g., using echo).

**\_\_invoke()**: Invoked when an object is called as a function.

**\_\_clone()**: Invoked when an object is cloned.

**\_\_debugInfo()**: Invoked when var\_dump() is called on the object to provide custom debug information.

* Static Keyword

The static keyword in PHP allows properties and methods to be shared among all instances of a class, rather than being tied to any specific instance. Static properties and methods can be accessed without creating an instance of the class.

Ex

<?php

class Counter {

// Static property to hold the count

public static $count = 0;

// Static method to increment the count

public static function increment() {

self::$count++; // Accessing static property using self::

}

// Static method to get the current count

public static function getCount() {

return self::$count;

}

}

// Accessing static properties and methods without creating an instance

Counter::increment();

Counter::increment();

echo "Current count: " . Counter::getCount(); // Output: Current count: 2

// Static properties are shared across all instances

$counter1 = new Counter();

$counter2 = new Counter();

$counter1::increment();

$counter2::increment();

echo "Current count after multiple increments: " . Counter::getCount(); // Output: Current count: 4

?>

11. Create multiple Traits and use it in to a single class?

**traits** are a mechanism for code reuse in single inheritance. A trait allows you to define methods that can be reused in multiple classes, thereby reducing duplication. A class can use multiple traits, which provides flexibility without the need for complex inheritance hierarchies.

Ex

<?php

// Define Trait 1

trait Logger {

public function log($message) {

echo "Logging message: $message\n";

}

}

// Define Trait 2

trait Notifier {

public function sendNotification($recipient, $message) {

echo "Sending notification to $recipient: $message\n";

}

}

// Define Trait 3

trait Greeter {

public function greet($name) {

echo "Hello, $name!\n";

}

}

// Class using multiple traits

class User {

use Logger, Notifier, Greeter; // Using multiple traits

public function createUser($name, $email) {

echo "Creating user: $name with email: $email\n";

$this->log("User $name created.");

$this->sendNotification($email, "Welcome $name!");

$this->greet($name);

}

}

// Instantiate the class and use its methods

$user = new User();

$user->createUser("Alice", "alice@example.com");

/\*

Output:

Creating user: Alice with email: alice@example.com

Logging message: User Alice created.

Sending notification to alice@example.com: Welcome Alice!

Hello, Alice!

\*/

?>

12. Use of The $this keyword?

The $this keyword in PHP is used to refer to the **current instance** of the class from which it is called. It is used within object methods to access the object's properties and methods. Essentially, $this is a pointer that allows you to refer to the object instance itself.

Ex

<?php

class User {

public $name;

public function setName($name) {

$this->name = $name; // Accessing the object's property using $this

}

public function getName() {

return $this->name; // Accessing the object's property using $this

}

}

$user = new User();

$user->setName("Alice");

echo $user->getName(); // Output: Alice

?>

JQuery:

1. What is jQuery?

jQuery is a fast, small, and feature-rich JavaScript library designed to simplify HTML document traversal and manipulation, event handling, and animation. It provides a user-friendly API that works across various browsers, allowing developers to write less code to achieve complex tasks.

**Key Features of jQuery:**

1. **DOM Manipulation**: Easily select and manipulate elements on a webpage, such as changing text, styles, or attributes.
2. **Event Handling**: Simplifies the process of attaching event handlers to elements, making it easier to manage user interactions like clicks and keyboard inputs.
3. **AJAX Support**: Provides methods for making asynchronous HTTP requests, allowing developers to load data from the server without refreshing the page.
4. **Animations and Effects**: Offers built-in methods to create animations and effects, such as fading elements in and out or sliding them up and down.
5. How are JavaScript and jQuery different?

|  |  |
| --- | --- |
| jQuery | JavaScript |
| A fast and feature-rich JavaScript library. | A high-level, interpreted programming language. |
| Simplified and concise. | More verbose & complex. |
| Easier with built-in methods. | Requires more code to achieve the same tasks. |
| Handles cross-browser issues internally. | Developers need to handle cross-browser compatibility manually. |
| Built-in methods for common animations. | Requires more complex code or use of CSS animations. |
| Extensive library of plugins available. | Requires custom code or third-party libraries. |
| Easier for beginners due to its simplicity. | Steeper learning curve. |

1. Which is the starting point of code execution in jQuery?

The starting point of code execution in jQuery is typically the $(document).ready() function. This function ensures that the code inside it runs only after the DOM (Document Object Model) is fully loaded and ready to be manipulated. This is important because it prevents errors that can occur when trying to interact with elements that haven't yet been rendered on the page.

Syntax:

$(document).ready(function() {

// Your jQuery code goes here

});