PHP Advanced OOP

Duration: 1 hours

Why Use Abstract Classes and Methods

Explanation:

Abstract classes are classes that cannot be instantiated on their own. They are used as base classes and can contain abstract methods, which are declared without an implementation.

Why Use Abstract Classes and Methods

- **Code Reusability**: Abstract classes allow you to define methods that must be implemented in any derived class, thus ensuring consistency while allowing flexibility.
- **Design Template**: They provide a blueprint for other classes and help in designing large-scale applications by enforcing a contract for subclasses.

Real-Life Example

- **User Authentication System**: Consider an abstract class **`User`** with abstract methods **`login`** and **`register`**. Different types of users like **`Admin`** and **`Customer`** can extend this class and implement these methods according to their specific needs.

```php
<?php
abstract class User {
    abstract public function login();
    abstract public function register();

    public function resetPassword() {
        // Common method for all types of users
    }
}
```

```php
class Admin extends User {
    public function login() {
        // Login logic for admin
    }

    public function register() {
        // Registration logic for admin
    }
}
```

```php
class Customer extends User {
    public function login() {
        // Login logic for customer
    }

    public function register() {
        // Registration logic for customer
    }
}
```

Dos:

- **Do Use for Base Classes**: Implement abstract classes as base classes where you have a general idea but want to leave the details to the subclasses.
- **Do Declare Abstract Methods**: Use abstract methods to enforce a contract that all subclasses must implement, ensuring consistency across various implementations.

Don'ts:

- **Don't Instantiate Directly**: Never try to instantiate an abstract class directly. It's meant to be extended by other classes.

- **Don't Overuse**: Avoid using abstract classes when a simple class or interface would suffice. Overuse can lead to an unnecessarily complicated class hierarchy.

Final Classes and Methods

Why Use Final Classes and Methods

- **Preventing Override**: To maintain the integrity of the class behavior, which is critical in certain situations.
- **Security and Stability**: Final classes/methods ensure that certain critical methods or classes remain unchanged and secure throughout the application lifecycle.

Dos:

- **Do Use to Prevent Overrides**: Use final classes and methods when you want to secure the core functionality of a method or the non-extendability of a class.
- **Do Apply for Critical Functions**: Apply final to methods that contain critical operations which should not be altered by any subclass, like security checks or payment processing.

Don'ts:

- **Don't Apply Arbitrarily**: Avoid using the final keyword without a clear reason. It restricts the extendability and flexibility of your classes.
- **Don't Confuse with Private**: Understand the difference between final and private; final methods can still be accessed by child classes but cannot be overridden.

Real-Life Example

- **Payment Processing System**: In a payment processing system, the method to process payments must be consistent and secure. Using `final` ensures that no subclass can alter this critical functionality.

```php
class PaymentProcessor {
    final public function processPayment($amount) {
        // Payment processing logic
    }
}

class CustomPaymentProcessor extends PaymentProcessor {
    // Cannot override processPayment
}
```

Interfaces

Why Use Interfaces

- **Enforcing Implementation**: Interfaces ensure that certain methods are implemented in the classes that implement them, which is crucial for consistency.
- **Multiple Inheritances**: PHP doesn't support multiple inheritances directly but through interfaces, a class can implement multiple interfaces.

Dos:

- **Do Implement for Multiple Inheritances**: Use interfaces when you need to implement multiple inheritances, as PHP doesn't support more than one class inheritance.
- **Do Use for Public Methods**: Define public methods in interfaces that you want the implementing classes to expose.

Don'ts:

- **Don't Include Implementation Details**: Avoid adding method implementations or non-public methods in interfaces. They should only declare method signatures.
- **Don't Over Interface**: While interfaces are useful, creating too many can lead to a complex and hard-to-maintain codebase. Use them judiciously.

Real-Life Example

- **API Integration**: For a web application integrating multiple APIs, an interface `APIIntegration` can declare methods like `connect`, `sendRequest`, and `receiveResponse`. Different classes for different API integrations can implement this interface.

```php
interface APIIntegration {
    public function connect();
    public function sendRequest($request);
    public function receiveResponse();
}

class TwitterAPI implements APIIntegration {
    public function connect() {
        // Connection logic for Twitter API
    }

    public function sendRequest($request) {
        // Send request to Twitter API
    }

    public function receiveResponse() {
        // Handle response from Twitter API
    }
}
```

The following table offers a comprehensive comparison, outlining key features, differences, and typical use cases for interfaces and abstract classes in PHP. This should provide a clearer understanding for anyone learning about these important object-oriented programming concepts.

Feature	Interface	Abstract Class
Code Composition	- Only abstract methods	- Abstract methods
	- Constants	- Constants
		- Concrete (implemented) methods
		- Concrete (instance) variables
Access Modifiers	- All methods are public by default	- Public, protected, and private methods

	- Cannot have private or protected methods	- Greater flexibility in access control
Inheritance/Implementation	- A class can implement multiple interfaces	- A class can extend only one abstract class
	- Used for defining a contract without implementation	- Can provide a common base with partial implementation
Typical Use Case	- Defining a contract that multiple classes can implement	- Providing a skeletal structure to subclasses
	- Useful in situations where multiple inheritance is needed	- Useful when subclasses have shared methods or fields
Insatiability	- Cannot be instantiated	- Cannot be instantiated directly
Extending/Implementing	- Other interfaces can extend interfaces (since PHP 8.0)	- Can be extended by other abstract or concrete classes

Inheritance

Why Use Inheritance

- **Code Reusability and Extension**: Allows for building new functionality on top of existing classes.
- **Hierarchical Structure**: Helps in organizing code in a hierarchical manner which makes it easier to manage and understand.

Real-Life Example

- **Content Management System (CMS)**: In a CMS, a base class `Content` might have properties like `title` and `body`, and methods like `save` and `display`. Subclasses such as `Article`, `BlogPost`, and `NewsItem` inherit from `Content` and can have additional properties or methods.

```php
class Content {
    protected $title;
    protected $body;

    public function save() {
        // Save content to database
    }

    public function display() {
        // Display content
    }
}

class Article extends Content {
    private $author;

    public function setAuthor($author) {
        $this->author = $author;
    }

    // Additional methods specific to Article
}
```

Dos:

- **Do Use for Related Classes**: Implement inheritance where classes are genuinely related and there is a clear hierarchical relationship.

- **Do Override Methods Judiciously**: Override methods to extend or modify functionality but maintain the general purpose and contract of the method.

Don'ts:

- **Don't Use for Unrelated Classes**: Avoid using inheritance just to reuse code. If two classes are not conceptually related, consider composition over inheritance.
- **Don't Overuse Deep Hierarchies**: Avoid creating very deep inheritance hierarchies as it can make the code difficult to follow and maintain.

Composition Vs Inheritance

- Composition implies a "has-a" relationship.
    - A **`Car`** class that contains objects of the **`Engine`** and **`Wheel`** classes. The **`Car`** "has an" **`Engine`** and "has" **`Wheels`**.
- Inheritance implies an "is-a" relationship.
    - A **`Bird`** class can be a subclass of an **`Animal`** class. The **`Bird`** "is an" **`Animal`** and inherits its characteristics, such as the ability to move or breathe.

Understanding self, parent, and Calling Parent Constructor in PHP

In PHP, **`self`**, **`parent`**, and the practice of calling a parent's constructor are fundamental concepts in object-oriented programming. They play a crucial role in class inheritance and behavior.

Using `self` Keyword

- **Purpose**: **`self`** refers to the same class in which it is used, particularly for accessing static properties and methods.
- **Scope**: It always refers to the class where it's written, not dynamically to any subclass that extends it.

```php
class Example {
    private static $instanceCount = 0;
    public function __construct() {
        self::$instanceCount++;
    }
    public static function getInstanceCount() {
        return self::$instanceCount;
    }
}
```

```php
class ExtendedExample extends Example {
    public $x = 10;
    public function __construct() {
        parent::__construct();
        print($this->x);
        print(self::getInstanceCount());
    }

}
```

Important Points : Parent Constructor

1. **Explicit Call**: In PHP, the parent class constructor is not called implicitly if the child class defines a constructor. You need to explicitly call it using `parent::__construct()`.
2. **Order of Execution**: Generally, you call the parent constructor at the beginning of the child constructor to ensure all initialization in the parent class is done before the child class's constructor logic. However, depending on your requirements, you can call it later in the child constructor.
3. **Passing Parameters**: If the parent constructor accepts parameters, you must pass them when calling `parent::__construct()`. For example:

```php
class ParentClass {
    public function __construct($param) {
        // Initialization code using $param
        echo "Parent constructor called with $param\n";
    }
}

class ChildClass extends ParentClass {
    public function __construct($param) {
        // Call parent constructor with parameter
        parent::__construct($param);

        // Additional initialization code
        echo "Child constructor called\n";
    }
}

$child = new ChildClass("test");
```