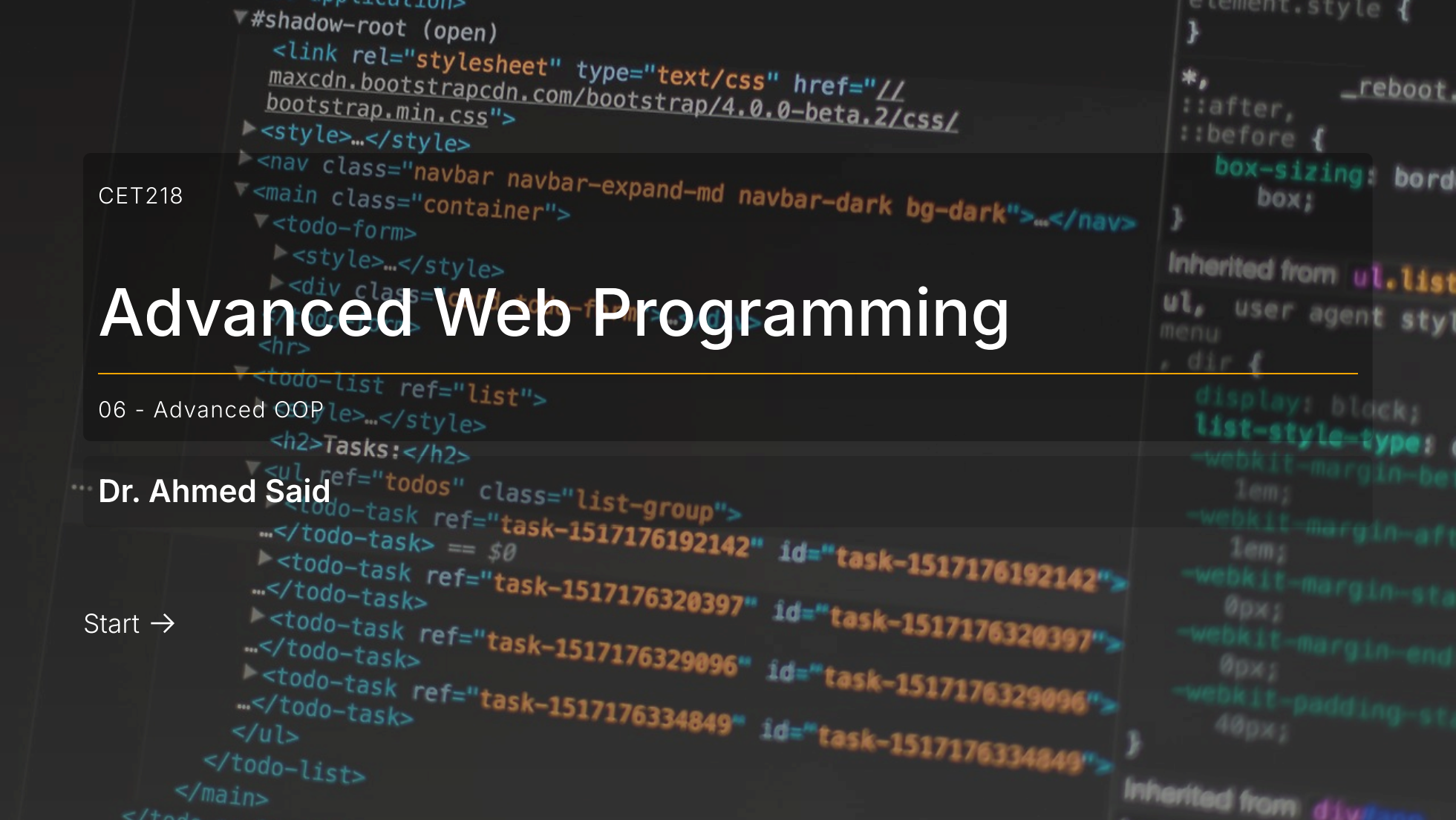
CET218

Advanced Web Programming

06 - Advanced OOP

Dr. Ahmed Said

Start →



Advanced PHP OOP Concepts

- We'll explore advanced features in PHP to create robust, maintainable, and scalable applications.
- Topics include:
  - Abstract Classes
  - Interfaces
  - Traits
  - Magic Methods (`__get` and `__set`)
  - Readonly Properties
  - Type Hinting
  - Serialization
  - MVC Pattern

Abstract Classes

- **Abstract classes** are blueprints that cannot be instantiated directly.
- They enforce a structure for subclasses by defining methods that must be implemented.
- Can include both **abstract methods** (no implementation) and **concrete methods** (with implementation).

```php
abstract class Shape {
    abstract public function area(); // Must be implemented by subclass
    public function description() { echo "This is a shape."; }
}

class Circle extends Shape {
    private $radius;
    public function __construct($radius) { $this->radius = $radius; }
    public function area() { return pi() * pow($this->radius, 2); }
}
$circle = new Circle(5);
echo $circle->area(); // Outputs: 78.5398...
$circle->description(); // Outputs: This is a shape.
```

Note: Abstract classes ensure subclasses adhere to a specific contract while sharing common functionality.

Interfaces

- **Interfaces** define a contract of methods that implementing classes must follow.
- No implementation is provided—only method signatures.
- Classes can implement multiple interfaces, enabling flexible designs.

```php
interface Logger {
    public function log($message);
}

class FileLogger implements Logger {
    public function log($message) {
        file_put_contents('log.txt', $message . PHP_EOL, FILE_APPEND);
    }
}

$logger = new FileLogger();
$logger->log("Error occurred"); // Writes to log.txt
```

Tip: Use interfaces to enforce consistent behavior across unrelated classes, promoting polymorphism.

Interfaces vs Abstract Classes in PHP

- Understanding the differences between interfaces and abstract classes is crucial for effective OOP design in PHP.

Interfaces vs Abstract Classes

- **Interface**
  - A **contract** that defines a set of methods a class must implement
  - Contains **only method signatures** and **constants**
  - No implementation provided
  - Classes can **implement multiple interfaces**
- **Abstract Class**
  - A class that **cannot be instantiated directly**
  - Can contain both **abstract methods** (no implementation) and **concrete methods** (with implementation)
  - Can have **properties** and **constructors**
  - Classes can **extend only one abstract class**

Interfaces vs Abstract Classes

Aspect	Interfaces	Abstract Classes
Multiple Inheritance	A class can implement multiple interfaces	A class can extend only one abstract class
Implementation	No implementation; only method signatures	Can provide partial implementation with concrete methods
Encapsulation	Only public methods	Can have protected and private members
Relationship Type	"Can-do" capability	"Is-a" relationship
Flexibility	More flexible for defining capabilities across different class hierarchies	Better for sharing code and state within a hierarchy

Examples

When to Use Interfaces

- Ideal for defining a **capability** that multiple unrelated classes can share
- Example: A `Logger` interface for different logging mechanisms

```php
interface Logger {
    public function log($message);
}

class FileLogger implements Logger {
    public function log($message) {
        // Log to file
        file_put_contents('log.txt', $message . PHP_EOL, FILE_APPEND);
    }
}

class DatabaseLogger implements Logger {
    public function log($message) {
        // Log to database
        // e.g., insert into a database table
    }
}
```

Examples

When to Use Abstract Classes

- Useful for providing **shared implementation** for a group of related classes
- Example: An abstract `Logger` class with a shared method

```php
abstract class Logger {
    protected function formatMessage($message) {
        return date('Y-m-d H:i:s') . ' - ' . $message;
    }
    abstract public function log($message);
}

class FileLogger extends Logger {
    public function log($message) {
        $formatted = $this->formatMessage($message);
        file_put_contents('log.txt', $formatted . PHP_EOL, FILE_APPEND);
    }
}
```

Dependency Injection (DI)

- **Dependency Injection (DI)** is a design pattern used in software development to achieve Inversion of Control (IoC) between classes and their dependencies.
  - Interfaces are ideal for dependency injection, promoting **loose coupling**
  - Example: Injecting a `Logger` interface into a class

```php
interface Logger {
    public function log($message);
}

function processData(Logger $logger, $data) {
    $logger->log("Processing: $data");
}
```

  - This allows swapping `FileLogger` or `DatabaseLogger` easily

Combining Interfaces and Abstract Classes

- Interfaces and abstract classes can work together for powerful abstractions.
- An abstract class can implement an interface and provide partial implementation.

```php
interface Printable {
    public function print();
}

abstract class Document implements Printable {
    protected $title;
    public function __construct($title) {
        $this->title = $title;
    }
}

class PDF extends Document {
    public function print() {
        echo "Printing PDF: $this->title";
    }
}

$pdf = new PDF("Report");
$pdf->print(); // Outputs: Printing PDF: Report
```

Traits

- PHP traits provide another way to share implementation across classes without inheritance
- Can serve as an alternative to abstract classes for code reuse

What Are Traits in PHP?

- Traits are a feature in PHP that allow **code reuse** in object-oriented programming, especially in a language that supports only single inheritance.
- They enable developers to share methods across multiple classes without relying on a parent-child inheritance relationship. You include a trait in a class using the `use` keyword.

```php
trait Loggable {
    public function log($message) {
        echo "Log: $message";
    }
}

class User {
    use Loggable;
}

$user = new User();
$user->log("User created"); // Outputs: Log: User created
```

In this case, the `User` class gains the `log()` method from the `Loggable` trait as if it were defined directly in the class.

Why Use Traits?

Traits offer several benefits:

- **Avoid Code Duplication**: Share methods across classes that don't share a common parent.
- **Bypass Single Inheritance**: PHP limits a class to one parent via `extends`, but a class can use multiple traits.
- **Modular Code**: Group related methods into reusable units.
- **Flexibility**: Add or remove functionality from classes easily.

For instance, if multiple classes need logging, you can define it once in a trait and reuse it.

How Traits Work

- When a class uses a trait, PHP **inserts the trait's methods into the class** as if they were written there. Traits can also:
  - Include **properties** (though this is discouraged due to potential conflicts).
  - Use other traits, enabling **trait composition**.

```php
trait A {
    public function methodA() { echo "Method A"; }
}
trait B {
    use A;
    public function methodB() { echo "Method B"; }
}

class MyClass { use B; }

$obj = new MyClass();
$obj->methodA(); // Outputs: Method A
$obj->methodB(); // Outputs: Method B
```

Here, `MyClass` gets both `methodA()` from trait `A` and `methodB()` from trait `B` via composition.

Resolving Method Conflicts

If a class and a trait define the same method, the **class method takes precedence**. If multiple traits define the same method, you must resolve the conflict using:

- `insteadof` : Specify which trait's method to use.
- `as` : Alias a method to a different name.

```php
trait A {
    public function sayHello() { echo "Hello from A"; }
}
trait B {
    public function sayHello() { echo "Hello from B"; }
}
class MyClass {
    use A, B {
        A::sayHello insteadof B; // Use A's sayHello
        B::sayHello as sayHelloB; // Alias B's sayHello
    }
}
$obj = new MyClass();
$obj->sayHello();   // Outputs: Hello from A
$obj->sayHelloB();  // Outputs: Hello from B
```

This ensures clarity when method names clash.

Traits vs. Inheritance

- **Traits**:
  - Enable **horizontal code reuse** across unrelated classes.
  - No "is-a" relationship; purely for sharing code.
  - Can be used in multiple classes freely.
- **Inheritance**:
  - Creates an "is-a" relationship (e.g., a `Dog` is an `Animal`).
  - Limited to one parent class in PHP.
  - Ideal for sharing state and behavior within a hierarchy.

Traits vs. Interfaces

- **Traits**:
  - Provide **implementation** (methods with bodies).
  - Can include properties (though not recommended).
- **Interfaces**:
  - Define a **contract** (method signatures only, no implementation).
  - Classes can implement multiple interfaces.

Tip: Use traits for reusable code and interfaces for defining what a class must do.

Potential Drawbacks

While traits are powerful, they have downsides:

- **Name Conflicts**: Multiple traits with the same method require manual resolution.
- **Tight Coupling**: Overuse can make classes overly dependent on traits.
- **Readability**: Excessive traits can obscure code flow.
- **Testing Complexity**: Traits can make unit testing harder if not isolated properly.

Use traits thoughtfully to avoid these issues.

Best Practices

To use traits effectively:

- **Keep Traits Small**: Focus on one feature or responsibility.
- **Avoid Properties**: Stick to methods to prevent state conflicts.
- **Document Clearly**: Explain each trait's purpose.
- **Resolve Conflicts Explicitly**: Use `insteadof` and `as` for clarity.
- **Complement Other Tools**: Pair traits with inheritance and interfaces, not as a replacement.

Real-World Examples

Traits shine in practical scenarios:

- **Logging**:

```php
trait Logger {
    public function log($msg) {
        echo "Logged: $msg";
    }
}

class Product {
    use Logger;
}
```

Real-World Examples

- **String Utilities**:

```php
trait StringUtils {
    public function slugify($string) {
        return strtolower(trim(preg_replace('/[^A-Za-z0-9-]+/', '-', $string)));
    }
}

class Post {
    use StringUtils;
    public function createSlug($title) {
        return $this->slugify($title);
    }
}

$post = new Post();
echo $post->createSlug("Hello World!"); // Outputs: hello-world
```

Other use cases include validation, event handling, or shared utility methods.

Magic Methods: `__get` and `__set`

- **Magic methods** are invoked automatically in specific scenarios.
- `__get($property)` handles reading inaccessible or undefined properties.
- `__set($property, $value)` handles writing to inaccessible or undefined properties.

```php
class Person {
    private $data = [];
    public function __get($property) {
        return $this->data[$property] ?? null;
    }
    public function __set($property, $value) {
        $this->data[$property] = $value;
    }
}

$person = new Person();
$person->name = "John"; // Calls __set
echo $person->name;     // Calls __get, outputs: John
```

Warning: Overuse of magic methods can obscure logic—use them sparingly.

Readonly Properties

- Introduced in PHP 8.1, **readonly properties** can only be set once (typically in the constructor).
- They enhance immutability by preventing changes after initialization.

```php
class ImmutableUser {
    public readonly string $name;
    public function __construct(string $name) {
        $this->name = $name;
    }
}

$user = new ImmutableUser("Alice");
echo $user->name; // Outputs: Alice
// $user->name = "Bob"; // Error: Cannot modify readonly property
```

Tip: Readonly properties are ideal for data that should remain constant after object creation.

Type Hinting

- **Type hinting** enforces the expected data types for parameters and return values.
- Supported for scalars (int, string), classes, interfaces, and more.
- Improves code reliability and readability.

```php
function add(int $a, int $b): int {
    return $a + $b;
}

echo add(5, 10); // Outputs: 15
// echo add(5, "10"); // Error: Argument 2 must be int
```

> **Note**: Type hinting catches type errors early, reducing runtime bugs.

Advanced Type Hinting

- PHP supports **union types** (PHP 8.0+) and **mixed** types (PHP 8.0+).
- **Union** types allow multiple acceptable types; **mixed** accepts any type.

```php
function processInput(string|int $input): string {
    return "Received: $input";
}

echo processInput(42);    // Outputs: Received: 42
echo processInput("test"); // Outputs: Received: test

function anything(mixed $value): void {
    var_dump($value);
}

anything(123); // Outputs: int(123)
anything("hi"); // Outputs: string(2) "hi"
```

Tip: Use advanced typing for flexibility without sacrificing safety.

Serialization

- **Serialization** converts objects into a string format for storage or transmission.
- Use `serialize()` to create the string and `unserialize()` to restore the object.
- Custom serialization can be defined with `__serialize()` and `__unserialize()`.

```php
class User {
    public $name;
    public function __construct($name) {
        $this->name = $name;
    }
}

$user = new User("Alice");
$serialized = serialize($user);
echo $serialized; // Outputs: O:4:"User":1:{s:4:"name";s:5:"Alice";}

$restored = unserialize($serialized);
echo $restored->name; // Outputs: Alice
```

> **Warning**: Avoid un-serializing untrusted data to prevent security vulnerabilities.

Custom Serialization

- `__serialize()` and `__unserialize()` allow precise control over serialization.

```php
class SecureUser {
    private $name;
    private $secret;
    public function __construct($name, $secret) {
        $this->name = $name;
        $this->secret = $secret;
    }
    public function __serialize() {
        return ['name' => $this->name]; // Exclude $secret
    }
    public function __unserialize(array $data) {
        $this->name = $data['name'];
        $this->secret = null; // Reset sensitive data
    }
}

$user = new SecureUser("Alice", "password");
$serialized = serialize($user);
$restored = unserialize($serialized);
```

Note: Custom serialization protects sensitive data during storage.

Polymorphism in PHP

Polymorphism in PHP

- **Polymorphism** is a core OOP concept that allows objects of different classes to be treated as objects of a common superclass or interface.
- It enables flexibility and reusability by allowing the same method to behave differently based on the object it's called on.
- In PHP, polymorphism is primarily achieved through:
  - **Method Overriding**: Subclasses provide their own implementation of a method defined in the parent class.
  - **Interfaces**: Classes implementing the same interface can be used interchangeably.

Types of Polymorphism

- There are two main types of polymorphism:
  - **Compile-time (Static) Polymorphism**: Resolved during compilation (e.g., function overloading). PHP does not support this directly.
  - **Runtime (Dynamic) Polymorphism**: Resolved at runtime, which is the focus in PHP (e.g., method overriding).
- PHP uses **runtime polymorphism** to determine which method to call based on the actual object type.

Polymorphism via Method Overriding

- **Method Overriding**: When a subclass provides its own implementation of a method that is already defined in its parent class.
- The overridden method in the subclass is called instead of the parent class method when invoked on a subclass object.

```php
class Animal {
    public function makeSound() {
        echo "Some generic sound";
    }
}

class Dog extends Animal {
    public function makeSound() {
        echo "Bark";
    }
}

$animal = new Animal();
$animal->makeSound(); // Outputs: Some generic sound
```

Polymorphism with Interfaces

- Interfaces allow unrelated classes to implement the same set of methods, enabling polymorphism.
- Objects of different classes can be treated uniformly if they implement the same interface.

```php
interface Shape {
    public function area();
}

class Circle implements Shape {
    private $radius;
    public function __construct($radius) {
        $this->radius = $radius;
    }
    public function area() {
        return pi() * pow($this->radius, 2);
    }
}

class Rectangle implements Shape {
    private $width, $height;
    public function __construct($width, $height) {
        $this->width = $width;
        $this->height = $height;
```

Polymorphism and Abstract Classes

- Abstract classes can define abstract methods that must be implemented by subclasses.
- This ensures that all subclasses provide their own implementation, enabling polymorphic behavior.

```php
abstract class Vehicle {
    abstract public function startEngine();
}

class Car extends Vehicle {
    public function startEngine() {
        echo "Car engine started";
    }
}

class Motorcycle extends Vehicle {
    public function startEngine() {
        echo "Motorcycle engine started";
    }
}

function startVehicleEngine(Vehicle $vehicle) {
    $vehicle->startEngine();
}
```

Type Hinting and Polymorphism

- **Type hinting** in function parameters ensures that only objects of a specific class or interface are passed.
- This is crucial for polymorphism, as it allows functions to accept any object that meets the type requirement.

```php
function describeAnimal(Animal $animal) {
    $animal->makeSound();
}

$dog = new Dog();
describeAnimal($dog); // Outputs: Bark
```

Tip: Type hinting with parent classes or interfaces enables polymorphic behavior while maintaining type safety.

Benefits of Polymorphism

- **Flexibility**: Write functions that work with objects of multiple types, as long as they share a common superclass or interface.
- **Code Reusability**: Reduce duplication by writing generic code that operates on a variety of object types.
- **Maintainability**: Easily extend the system by adding new classes without modifying existing code.

> **Note**: Polymorphism is a powerful tool for creating scalable and adaptable applications.