# HiLCoE School of Computer Science & Technology

## Chapter Three: Advanced Topic In PHP

**Course Title : Web Technologies II**

**Instructor name**: Yitayew Solomon
**E-mail address**: yitayewsolomon3@gmail.com

# Access Modifiers in PHP

## 3. Access Modifiers

Access modifiers determine the visibility of properties and methods.

- **public**: Accessible from anywhere.
- **private**: Accessible only within the class itself.
- **protected**: Accessible within the class and by inherited classes.

# Cont. ...

**Access Modifiers** in PHP are keywords that define the visibility or scope of class properties and methods. They control how and where the properties and methods of a class can be accessed—whether inside the class, outside the class, or in child classes (in case of inheritance). The three primary access modifiers in PHP are `public`, `private`, and `protected`.

---

## 1. Public Access Modifier

- `public` is the most open access modifier, allowing the class's properties and methods to be accessed from anywhere in the code.
- If a property or method is declared as `public`, it can be accessed from:
  - Within the class itself.
  - Outside the class (from other parts of the program).
  - In child classes (if the class is inherited).

# Cont. ...

C:\xampp\htdocs\web\public_AM.php • - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

public_AM.php

```php
<?php
class Car {
    public $make;
    public $model;

    public function setDetails($make, $model) {
        $this->make = $make;
        $this->model = $model;
    }

    public function getDetails() {
        return "This car is a " . $this->make . " " . $this->model;
    }
}

// Creating an object and accessing public properties and methods
$car = new Car();
$car->setDetails("Honda", "Civic");
echo $car->getDetails();  // Output: This car is a Honda Civic
?>
```

# Private Accesses Modifier

## 2. Private Access Modifier

- `private` restricts access to properties and methods so that they are only accessible **within the class itself**.
- If a property or method is declared as `private`, it **cannot** be accessed:
  - From outside the class.
  - In child classes (even if the class is inherited).
- This is useful for encapsulating sensitive information or internal logic that shouldn't be exposed outside the class.

**Example:**

# Cont. ...

C:\xampp\htdocs\web\Private_AM.php • - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Private_AM.php

```php
<?php
class BankAccount {
    private $balance;

    public function __construct($initialBalance) {
        $this->balance = $initialBalance;
    }

    public function deposit($amount) {
        $this->balance += $amount;
    }

    public function getBalance() {
        return "Current balance: $" . $this->balance;
    }
}

$account = new BankAccount(100);
// $account->balance = 200; // This will throw an error as balance is private
$account->deposit(50);
echo $account->getBalance();  // Output: Current balance: $150
?>
```

# Protected Accesses Modifier

## 3. Protected Access Modifier

- `protected` allows properties and methods to be accessed within the class itself and by classes that inherit from that class.
- However, like `private`, protected members cannot be accessed from outside the class.
- The main use of `protected` is in inheritance, where it allows child classes to use or override the properties and methods of the parent class.

**Example:**

# Example

C:\xampp\htdocs\web\Protected_AM.php - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Protected_AM.php

```php
<?php
class Vehicle {
    protected $fuelType;

    public function setFuelType($fuel) {
        $this->fuelType = $fuel;
    }

    protected function getFuelType() {
        return $this->fuelType;
    }
}

class Car extends Vehicle {
    public function showFuelType() {
        return "This car runs on " . $this->getFuelType();
    }
}

$car = new Car();
$car->setFuelType("Gasoline");
echo $car->showFuelType();  // Output: This car runs on Gasoline
?>
```

# Cont. ...

## 4. Summary of Access Modifiers

| Modifier | Accessibility Within Class | Accessibility Outside Class | Accessibility in Child Classes |
|---|---|---|---|
| public | Yes | Yes | Yes |
| private | Yes | No | No |
| protected | Yes | No | Yes |

# Cont. ...

## 5. Importance of Access Modifiers in OOP

- **Encapsulation**: Access modifiers play a critical role in encapsulating the internal workings of a class. By restricting access to certain properties and methods, classes can hide sensitive data and expose only what is necessary.
- **Security**: Limiting access to sensitive or critical properties (e.g., user passwords or financial data) using `private` and `protected` prevents unauthorized manipulation from outside the class.
- **Inheritance**: With `protected`, developers can allow child classes to inherit and use certain properties and methods, while still preventing direct access from outside.
- **Code Maintenance**: By using access modifiers, developers can maintain a clean and clear interface for their classes. Only the relevant methods and properties are exposed to users, making the code easier to understand and manage.

# Exercise

Create a `BankAccount` class that models a basic bank account. This class should have:

1. **Properties:**
   - A `public` property called `$accountNumber`.
   - A `protected` property called `$balance`.
   - A `private` property called `$pin`.
2. **Methods:**
   - A `public` method `deposit($amount)` that allows depositing money into the account by increasing `$balance`.
   - A `protected` method `getBalance()` that returns the current balance (only accessible within the class or subclasses).

# Cont. ...

- A `public` method `showBalance($enteredPin)` that takes an entered PIN as an argument, checks if it matches the `$pin`, and if it does, displays the balance using the `getBalance()` method. If the PIN does not match, it should display an error message.

3. **Constructor:**

- A constructor that takes `$accountNumber`, `$initialBalance`, and `$pin` as arguments and sets them to the appropriate properties.

Write the code for this class, create an object of the class, and demonstrate depositing an amount and viewing the balance using the correct PIN.

# Inheritance

## 1. Concept of Inheritance

Inheritance allows the child class to gain access to the attributes and behaviors defined in the parent class without needing to duplicate the code. This means that the child class can extend or modify the behavior of the parent class while still using its existing properties and methods.

**Benefits of Inheritance:**

- **Code Reusability**: Developers can reuse existing code, which reduces redundancy.
- **Logical Hierarchy**: Inheritance helps model real-world relationships in a structured way, making it easier to understand the code.
- **Easy Maintenance**: Changes made in the parent class automatically propagate to the child classes, making the code easier to maintain.

# Cont. ...

## 2. Types of Inheritance in PHP

1. **Single Inheritance:** A class can inherit from one parent class only. PHP does not support multiple inheritance, meaning a class cannot inherit from multiple classes.
2. **Multilevel Inheritance:** A class can inherit from another class, which in turn can inherit from a different class, creating a chain of inheritance.
3. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.

**Example of Single Inheritance:**

# Single Inheritance

php

Copy code

```php
<?php
class Animal {
    public function sound() {
        return "Animal makes a sound";
    }
}

class Dog extends Animal {
    public function sound() {
        return "Dog barks";
    }
}

$dog = new Dog();
echo $dog->sound(); // Output: Dog barks
?>
```

# Cont. ...

## 3. Key Components of Inheritance

- **Parent Class:** The class that provides properties and methods to the child class.
- **Child Class:** The class that inherits the properties and methods of the parent class. It can also have its own properties and methods or override inherited methods.

### Example of Multilevel Inheritance:

Here:

- `Vehicle` is the parent class.
- `Car` is a child class that inherits from `Vehicle`.
- `ElectricCar` is a child class that inherits from `Car`.

# Cont. ...

C:\xampp\htdocs\web\Multiple_Inheritance.php • - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Multiple_Inheritance.php

```php
<?php
class Vehicle {
    public function type() {
        return "Vehicle";
    }
}

class Car extends Vehicle {
    public function model() {
        return "Car Model";
    }
}

class ElectricCar extends Car {
    public function fuel() {
        return "Electric";
    }
}

$myElectricCar = new ElectricCar();
echo $myElectricCar->type(); // Output: Vehicle
echo $myElectricCar->model(); // Output: Car Model
echo $myElectricCar->fuel(); // Output: Electric
?>
```

# Overriding Methods

## 4. Overriding Methods

Child classes can override methods defined in the parent class. This means that the child class can provide its own implementation of a method that is already defined in the parent class.

**Example of Method Overriding:**

# Cont. ...

C:\xampp\htdocs\web\Method_Overriding.php - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Single_Inhertance.php | Method_Overriding.php

```php
<?php
class Shape {
    public function area() {
        return "Calculating area";
    }
}

class Circle extends Shape {
    public function area() {
        return "Calculating area of Circle";
    }
}

$circle = new Circle();
echo $circle->area(); // Output: Calculating area of Circle
?>
```

# Cont. ...

## 5. Accessing Parent Class Methods and Properties

Child classes can access the parent class's properties and methods using the `parent` keyword. This is particularly useful when you want to extend the behavior of a parent method while still using its functionality.

**Example:**

Here, the `Employee` class extends the functionality of the `greet()` method by calling the parent method and appending additional information.

# Cont. ...

C:\xampp\htdocs\web\Accessesing_Parent_class.php - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Accessesing_Parent_class.php

```php
<?php
class Person {
    public function greet() {
        return "Hello!";
    }
}

class Employee extends Person {
    public function greet() {
        return parent::greet() . " I am an employee.";
    }
}

$employee = new Employee();
echo $employee->greet(); // Output: Hello! I am an employee.
?>
```

# Exercise

## Exercise

Create a `Vehicle` class and a `Car` class that inherits from `Vehicle`. Implement basic inheritance by doing the following:

1. **Vehicle Class:**
   - Add a `public` property `$make` for the vehicle's make (e.g., Toyota).
   - Add a `public` property `$year` for the vehicle's manufacturing year.
   - Add a `public` method `displayInfo()` that displays the make and year of the vehicle.
2. **Car Class** (inherits from `Vehicle`):
   - Add a `public` property `$doors` to store the number of doors (e.g., 2 or 4).
   - Add a `public` method `displayCarInfo()` that calls `displayInfo()` from the `Vehicle` class and also displays the number of doors.
3. Create an instance of the `Car` class, set values for the make, year, and doors, and then call `displayCarInfo()` to see the output. ↓

# Polymorphism

## Polymorphism in PHP: Theoretical Explanation

**Polymorphism** is a fundamental concept in Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. The term "polymorphism" comes from the Greek words "poly," meaning many, and "morph," meaning forms. In the context of PHP and other OOP languages, polymorphism enables a single interface to represent different underlying forms (data types).

---

# Cont. ...

## 1. Types of Polymorphism

Polymorphism can be classified into two main types:

1. **Compile-Time Polymorphism** (also known as static polymorphism): This type is resolved during compile time. In PHP, this can be achieved through **method overloading**, although PHP does not support method overloading in the traditional sense like some other languages. Instead, similar behavior can be implemented using default arguments.
2. **Run-Time Polymorphism** (also known as dynamic polymorphism): This type is resolved during runtime and is typically achieved through **method overriding** in child classes. This allows different classes to provide different implementations of the same method defined in a parent class.

# Cont. ...

## 2. Method Overriding

**Method overriding** is a common way to implement run-time polymorphism. In this scenario, a child class provides a specific implementation of a method that is already defined in its parent class. When the method is called on an object of the child class, the overridden method in the child class is executed, even if the object is referred to by a parent class reference.

**Example of Method Overriding:**

# Cont. ...

C:\xampp\htdocs\web\Overriding.php - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Overriding.php

```php
<?php
class Animal {
    public function sound() {
        return "Some sound";
    }
}

class Dog extends Animal {
    public function sound() {
        return "Bark";
    }
}

class Cat extends Animal {
    public function sound() {
        return "Meow";
    }
}

function animalSound(Animal $animal) {
    echo $animal->sound() . PHP_EOL;
}

$dog = new Dog();
$cat = new Cat();

animalSound($dog); // Output: Bark
animalSound($cat); // Output: Meow
?>
```

# Cont. ...

In this example:

- Both `Dog` and `Cat` classes override the `sound()` method defined in the `Animal` class.
- The `animalSound()` function accepts an `Animal` type reference, allowing it to handle any object derived from `Animal`. The correct `sound()` method is called based on the actual object type (either `Dog` or `Cat`).

# Cont. ...

## 3. Interfaces and Polymorphism

Another way to achieve polymorphism in PHP is through the use of **interfaces**. An interface defines a contract that implementing classes must adhere to, allowing for multiple classes to implement the same interface in different ways.

**Example Using Interfaces:**

# Cont. ...

C:\xampp\htdocs\web\Interfaces.php - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Interfaces.php

```php
<?php
interface Shape {
    public function area();
}
class Rectangle implements Shape {
    private $width;
    private $height;

    public function __construct($width, $height) {
        $this->width = $width;
        $this->height = $height;
    }

    public function area() {
        return $this->width * $this->height;
    }
}
class Circle implements Shape {
    private $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }

```

# Cont. ...

C:\xampp\htdocs\web\Interfaces.php - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Interfaces.php

```php
    public function area() {
        return pi() * ($this->radius ** 2);
    }
}
function calculateArea(Shape $shape) {
    echo "Area: " . $shape->area() . PHP_EOL;
}

$rectangle = new Rectangle(10, 5);
$circle = new Circle(3);

calculateArea($rectangle); // Output: Area: 50
calculateArea($circle); // Output: Area: 28.274333882308
?>
```

# Cont. ...

In this example:

- The `Shape` interface declares a method `area()`.
- Both `Rectangle` and `Circle` implement the `Shape` interface, providing their own definitions for the `area()` method.
- The `calculateArea()` function can accept any object that implements the `Shape` interface, demonstrating polymorphism.

---

# Cont. ...

## 4. Benefits of Polymorphism

- **Flexibility**: Polymorphism allows for flexibility in code, as different classes can be treated as instances of the same class or interface.
- **Code Reusability**: It promotes code reuse, as functions can operate on base class types, allowing different derived types to be passed in.
- **Maintenance**: Polymorphism facilitates easier code maintenance and extension. New classes can be added without changing existing code, as long as they adhere to the same interface or base class.

---

# Abstraction in PHP

**Abstraction** is a fundamental concept in Object-Oriented Programming (OOP) that focuses on exposing only the essential features of an object while hiding the complex implementation details. It allows developers to define a clear interface for interacting with objects, making it easier to work with complex systems. In PHP, abstraction can be achieved using **abstract classes** and **interfaces.**

# Cont. ...

## 1. Key Concepts of Abstraction

- **Hiding Complexity**: Abstraction helps in reducing complexity by hiding the internal workings of a class. Users of the class need only to understand its interface, not the implementation.
- **Defining Interfaces**: By defining a clear interface for an object, abstraction ensures that the essential properties and behaviors of the object are accessible, while the underlying complexity is concealed.
- **Improving Code Maintainability**: By separating the interface from the implementation, abstraction improves code maintainability. Changes to the implementation do not affect code that relies on the interface.

---

# Cont. ...

## 2. Abstract Classes

An **abstract class** in PHP is a class that cannot be instantiated on its own and may contain abstract methods (methods without implementation) that must be implemented by its derived classes. This is useful when you want to provide a common base class for a group of related classes.

### Example of Abstract Class:

In this example:

- The `Animal` class is abstract and contains an abstract method `sound()`.
- The `Dog` and `Cat` classes extend the `Animal` class and provide their own implementations of the `sound()` method.
- The `eat()` method is a regular method that can be used by all subclasses.

# Cont. ...

## 3. Interfaces

An **interface** defines a contract that classes must follow. An interface can contain abstract methods, which means that any class that implements the interface must provide an implementation for those methods. Unlike abstract classes, an interface cannot have any implementation, only method declarations.

**Example of Interface:**

In this example:

- The `Shape` interface defines a contract with the `area()` method.
- Both the `Rectangle` and `Circle` classes implement the `Shape` interface, providing their own implementation of the `area()` method.

# Cont. ...

C:\xampp\htdocs\web\Abstract.php - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Interfaces.php | Abstract.php

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

class Rectangle implements Shape {
    private $width;
    private $height;

    public function __construct($width, $height) {
        $this->width = $width;
        $this->height = $height;
    }

    public function area() {
        return $this->width * $this->height;
    }
}

```

# Cont. ...

C:\xampp\htdocs\web\Abstract.php • - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Interfaces.php | Abstract.php

```php
class Circle implements Shape {
    private $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }

    public function area() {
        return pi() * ($this->radius ** 2);
    }
}

$rectangle = new Rectangle(10, 5);
$circle = new Circle(3);

echo "Rectangle area: " . $rectangle->area() . PHP_EOL; // Output: Rectangle area: 50
echo "Circle area: " . $circle->area() . PHP_EOL;       // Output: Circle area: 28.274333882308
?>
```

# Cont. ...

## 4. Benefits of Abstraction

- **Reduced Complexity**: By hiding implementation details, abstraction simplifies the interface of complex systems.
- **Enhanced Code Organization**: Abstraction leads to better organization of code by separating interface from implementation.
- **Improved Code Reusability**: Common functionalities can be defined in abstract classes or interfaces, making it easier to reuse code across different parts of the application.
- **Easier Maintenance and Extensibility**: Changes in implementation can be made without affecting the code that relies on the interface, improving maintainability.

# Encapsulation in PHP

**Encapsulation** is a fundamental concept in Object-Oriented Programming (OOP) that involves bundling the data (attributes) and methods (functions) that operate on that data into a single unit called a class. It restricts direct access to some of an object's components, which is a means of preventing unintended interference and misuse of the methods and data. Encapsulation helps in protecting the integrity of the object's state and provides a controlled way to access and modify it.

# Cont. ...

## 1. Key Concepts of Encapsulation

- **Data Hiding**: Encapsulation promotes the concept of data hiding, where the internal state of an object is kept private and can only be accessed or modified through public methods. This protects the object's state from unintended changes and maintains data integrity.
- **Public, Private, and Protected Modifiers**:
  - **Public**: Properties and methods declared as public can be accessed from anywhere.
  - **Private**: Properties and methods declared as private can only be accessed within the class itself. They are hidden from any outside access.
  - **Protected**: Properties and methods declared as protected can be accessed within the class and by derived classes.
- **Getters and Setters**: To provide controlled access to the private properties, getter and setter methods are often used. Getters retrieve the value of a property, while setters allow modification of that property.

# Cont. ...

C:\xampp\htdocs\web\Encapsulation.php • - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Encapsulation.php

```php
<?php
class BankAccount {
    // Private properties
    private $accountNumber;
    private $balance;
    // Constructor to initialize the properties
    public function __construct($accountNumber, $initialBalance) {
        $this->accountNumber = $accountNumber;
        $this->balance = $initialBalance;
    }

    // Getter for accountNumber
    public function getAccountNumber() {
        return $this->accountNumber;
    }
    // Getter for balance
    public function getBalance() {
        return $this->balance;
    }

    // Method to deposit money
    public function deposit($amount) {
        if ($amount > 0) {
            $this->balance += $amount;
            return true;
        }
        return false;
    }
```

# Cont. ...

C:\xampp\htdocs\web\Encapsulation.php • - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

Encapsulation.php

```php

    // Method to withdraw money
    public function withdraw($amount) {
        if ($amount > 0 && $this->balance >= $amount) {
            $this->balance -= $amount;
            return true;
        }
        return false;
    }
}

// Example usage
$account = new BankAccount("123456789", 1000);
echo "Account Number: " . $account->getAccountNumber() . PHP_EOL; // Output: Account Number: 123456789
echo "Initial Balance: " . $account->getBalance() . PHP_EOL;      // Output: Initial Balance: 1000

$account->deposit(500);
echo "Balance after deposit: " . $account->getBalance() . PHP_EOL; // Output: Balance after deposit: 1500

if ($account->withdraw(300)) {
    echo "Withdrawal successful. New Balance: " . $account->getBalance() . PHP_EOL; // Output: Withdrawal
    successful. New Balance: 1200
} else {
    echo "Insufficient funds for withdrawal." . PHP_EOL;
}
?>
```

# Cont. ...

In this example:

- The `BankAccount` class encapsulates the `accountNumber` and `balance` properties as private, meaning they cannot be accessed directly from outside the class.
- The constructor initializes these properties when a new object is created.
- Getters (`getAccountNumber()` and `getBalance()`) provide controlled access to the private properties, allowing other classes or functions to read these values without modifying them.
- The `deposit()` and `withdraw()` methods allow controlled modification of the `balance`, ensuring that only valid operations can be performed.

---

# Cont. ...

### 3. Benefits of Encapsulation

- **Data Integrity**: By restricting direct access to properties, encapsulation helps maintain the integrity of the object's state, preventing invalid data from being set.
- **Improved Maintainability**: Encapsulation allows for changes in the internal implementation of a class without affecting external code that uses it, leading to easier maintenance and refactoring.
- **Controlled Access**: Getters and setters provide a way to control how properties are accessed and modified, allowing for additional logic (e.g., validation) to be implemented when properties are set.
- **Enhanced Security**: Encapsulation protects sensitive data by hiding it from direct access, which is crucial in scenarios like financial applications.

# Thank you!

Appreciate your action.