Object Oriented Programming - PHP

- ```
  <?php
  class Fruit {
    // code goes here...
  }
  ?>
  ```
- ```
  <?php
  class Fruit {
    // Properties
    public $name;
    public $color;

    // Methods
    function set_name($name) {
      $this->name = $name;
    }
    function get_name() {
      return $this->name;
    }
  }
  ?>
  ```

- ```
  <?php
  class Fruit {
    // Properties
    public $name;
    public $color;

    // Methods
    function set_name($name) {
      $this->name = $name;
    }
    function get_name() {
      return $this->name;
    }
  }

  $apple = new Fruit();
  $banana = new Fruit();
  $apple->set_name('Apple');
  $banana->set_name('Banana');

  echo $apple->get_name();
  echo "<br>";
  echo $banana->get_name();
  ?>
  ```
  ```

- ```
  <?php
  class Fruit {
    // Properties
    public $name;
    public $color;

    // Methods
    function set_name($name) {
      $this->name = $name;
    }
    function get_name() {
      return $this->name;
    }
    function set_color($color) {
      $this->color = $color;
    }
    function get_color() {
      return $this->color;
    }
  }

  $apple = new Fruit();
  $apple->set_name('Apple');
  $apple->set_color('Red');
  echo "Name: " . $apple->get_name();
  echo "<br>";
  echo "Color: " . $apple->get_color();
  ?>
  ```
  ```

- The $this keyword refers to the current object, and is only available inside methods.
- ```
  <?php
  class Fruit {
    public $name;
    function set_name($name) {
      $this->name = $name;
    }
  }
  $apple = new Fruit();
  $apple->set_name("Apple");
  ?>
  ```
- ```
  <?php
  class Fruit {
    public $name;
  }
  $apple = new Fruit();
  $apple->name = "Apple";
  ?>
  ```

- <?php
  $apple = new Fruit();
  var_dump($apple instanceof Fruit);
  ?>
- bool(true)

- If you create a __construct() function, PHP will automatically call this function when you create an object from a class.
- The construct function starts with two underscores (__)!
- ```
  <?php
  class Fruit {
    public $name;
    public $color;

    function __construct($name) {
      $this->name = $name;
    }
    function get_name() {
      return $this->name;
    }
  }

  $apple = new Fruit("Apple");
  echo $apple->get_name();
  ?>
  ```
  ```

- If you create a __destruct() function, PHP will automatically call this function at the end of the script.
- The destruct function starts with two underscores (__)!
- <?php
  class Fruit {
    public $name;
    public $color;

    function __construct($name) {
      $this->name = $name;
    }
    function __destruct() {
      echo "The fruit is {$this->name}.";
    }
  }

  $apple = new Fruit("Apple");
  ?>

- Properties and methods can have access modifiers which control where they can be accessed.
- There are three access modifiers:
  - public - the property or method can be accessed from everywhere. This is default
  - protected - the property or method can be accessed within the class and by classes derived from that class
  - private - the property or method can ONLY be accessed within the class
- ```
  <?php
  class Fruit {
    public $name;
    protected $color;
    private $weight;
  }

  $mango = new Fruit();
  $mango->name = 'Mango'; // OK
  $mango->color = 'Yellow'; // ERROR
  $mango->weight = '300'; // ERROR
  ?>
  ```
  ```

- ```
  <?php
  class Fruit {
    public $name;
    public $color;
    public $weight;

    function set_name($n) {  // a public function (default)
      $this->name = $n;
    }
    protected function set_color($n) { // a protected function
      $this->color = $n;
    }
    private function set_weight($n) { // a private function
      $this->weight = $n;
    }
  }

  $mango = new Fruit();
  $mango->set_name('Mango'); // OK
  $mango->set_color('Yellow'); // ERROR
  $mango->set_weight('300'); // ERROR
  ?>
  ```
  ```

- Inheritance in OOP = When a class derives from another class.
- The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods.
- An inherited class is defined by using the extends keyword.
- ```
  <?php
  class Fruit {
    public $name;
    public $color;
    public function __construct($name, $color) {
      $this->name = $name;
      $this->color = $color;
    }
    public function intro() {
      echo "The fruit is {$this->name} and the color is {$this->color}.";
    }
  }
  // Strawberry is inherited from Fruit
  class Strawberry extends Fruit {
    public function message() {
      echo "Am I a fruit or a berry? ";
    }
  }
  $strawberry = new Strawberry("Strawberry", "red");
  $strawberry->message();
  $strawberry->intro();
  ?>
  ```
  ```

```php
<?php
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  protected function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}
class Strawberry extends Fruit {
  public function message() {
    echo "Am I a fruit or a berry? ";
  }
}
// Try to call all three methods from outside class
$strawberry = new Strawberry("Strawberry", "red");  // OK. __construct() is public
$strawberry->message(); // OK. message() is public
$strawberry->intro(); // ERROR. intro() is protected
?>
```

- ```
  <?php
  class Fruit {
    public $name;
    public $color;
    public function __construct($name, $color) {
      $this->name = $name;
      $this->color = $color;
    }
    protected function intro() {
      echo "The fruit is {$this->name} and the color is {$this->color}.";
    }
  }
  class Strawberry extends Fruit {
    public function message() {
      echo "Am I a fruit or a berry? ";
      // Call protected method from within derived class - OK
      $this -> intro();
    }
  }
  $strawberry = new Strawberry("Strawberry", "red"); // OK.
  __construct() is public
  $strawberry->message(); // OK. message() is public and it calls
  intro() (which is protected) from within the derived class
  ?>
  ```
  ```

- Inherited methods can be overridden by redefining the methods (use the same name) in the child class.
-
```
<?php
class Fruit {
  public $name;
  public $color;
  public function __construct($name, $color) {
    $this->name = $name;
    $this->color = $color;
  }
  public function intro() {
    echo "The fruit is {$this->name} and the color is {$this->color}.";
  }
}
class Strawberry extends Fruit {
  public $weight;
  public function __construct($name, $color, $weight) {
    $this->name = $name;
    $this->color = $color;
    $this->weight = $weight;
  }
  public function intro() {
    echo "The fruit is {$this->name}, the color is {$this->color}, and the weight is {$this-
>weight} gram.";
  }
}
$strawberry = new Strawberry("Strawberry", "red", 50);
$strawberry->intro();
?>
```

- The final keyword can be used to prevent class inheritance or to prevent method overriding.
- ```
  <?php
  final class Fruit {
    // some code
  }
  // will result in error
  class Strawberry extends Fruit {
    // some code
  }
  ?>
  ```
- ```
  <?php
  class Fruit {
    final public function intro() {
      // some code
    }
  }
  class Strawberry extends Fruit {
    // will result in error
    public function intro() {
      // some code
    }
  }
  ?>
  ```

- Constants cannot be changed once it is declared.
- Class constants can be useful if you need to define some constant data within a class.
- A class constant is declared inside a class with the const keyword.
- Class constants are case-sensitive. However, it is recommended to name the constants in all uppercase letters.
- Can access a constant from outside the class by using the class name followed by the scope resolution operator (::) followed by the constant name.

- ```
  <?php
  class Goodbye {
    const LEAVING_MESSAGE = "Thank you";
  }

  echo Goodbye::LEAVING_MESSAGE;
  ?>
  ```
  ```

- Can access a constant from inside the class by using the ***self*** keyword followed by the scope resolution operator ***(::)*** followed by the constant name.
- ```
  <?php
  class Goodbye {
    const LEAVING_MESSAGE = "Thank you";
    public function byebye() {
      echo self::LEAVING_MESSAGE;
    }
  }
  $goodbye = new Goodbye();
  $goodbye->byebye();
  ?>
  ```
  ```

- Abstract classes and methods are when the parent class has a named method, but need its child class(es) to fill out the tasks.
- An abstract class is a class that contains at least one abstract method. An abstract method is a method that is declared, but not implemented in the code.
- An abstract class or method is defined with the abstract keyword:
- <?php
  abstract class ParentClass {
    abstract public function someMethod1();
    abstract public function someMethod2($name, $color);
    abstract public function someMethod3() : string;
  }
  ?>

- So, when a child class is inherited from an abstract class, we have the following rules:
  - The child class method must be defined with the same name and it redeclares the parent abstract method
  - The child class method must be defined with the same or a less restricted access modifier
  - The number of required arguments must be the same. However, the child class may have optional arguments in addition

```php
<?php
// Parent class
abstract class Car {
  public $name;
  public function __construct($name) {
    $this->name = $name;
  }
  abstract public function intro();
}
// Child classes
class Audi extends Car {
  public function intro() {
    return "Choose German quality! I'm an $this->name!";
  }
}
class Volvo extends Car {
  public function intro() {
    return "Proud to be Swedish! I'm a $this->name!";
  }
}
class Citroen extends Car {
  public function intro() {
    return "French extravagance! I'm a $this->name!";
  }
}
// Create objects from the child classes
$audi = new audi("Audi");
echo $audi->intro(); echo "<br>";
$volvo = new volvo("Volvo");
echo $volvo->intro(); echo "<br>";
$citroen = new citroen("Citroen");
echo $citroen->intro();
?>
```

- ```
  <?php
  abstract class ParentClass {
    // Abstract method with an argument
    abstract protected function prefixName($name);
  }

  class ChildClass extends ParentClass {
    public function prefixName($name) {
      if ($name == "John Doe") {
        $prefix = "Mr.";
      } elseif ($name == "Jane Doe") {
        $prefix = "Mrs.";
      } else {
        $prefix = "";
      }
      return "{$prefix} {$name}";
    }
  }

  $class = new ChildClass;
  echo $class->prefixName("John Doe");
  echo "<br>";
  echo $class->prefixName("Jane Doe");
  ?>
  ```
  ```

- PHP only supports single inheritance: a child class can inherit only from one single parent.
- So, what if a class needs to inherit multiple behaviors? OOP traits solve this problem.
- Traits are used to declare methods that can be used in multiple classes. 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 are declared with the trait keyword

- ```
  <?php
  trait TraitName {
    // some code...
  }
  ?>
  ```
- ```
  <?php
  class MyClass {
    use TraitName;
  }
  ?>
  ```

- ```
  <?php
  trait message1 {
  public function msg1() {
      echo "OOP is fun! ";
    }
  }

  class Welcome {
    use message1;
  }

  $obj = new Welcome();
  $obj->msg1();
  ?>
  ```
  ```

- ```
  <?php
  trait message1 {
    public function msg1() {
      echo "OOP is fun! ";
    }
  }
  trait message2 {
    public function msg2() {
      echo "OOP reduces code duplication!";
    }
  }
  class Welcome {
    use message1;
  }
  class Welcome2 {
    use message1, message2;
  }
  $obj = new Welcome();
  $obj->msg1();
  echo "<br>";
  $obj2 = new Welcome2();
  $obj2->msg1();
  $obj2->msg2();
  ?>
  ```
  ```

- Static methods can be called directly - without creating an instance of a class.
- Static methods are declared with the static keyword
-
```
<?php
class ClassName {
  public static function staticMethod() {
    echo "Hello World!";
  }
}
?>
```
- ClassName::staticMethod();

- ```
  <?php
  class greeting {
    public static function welcome() {
      echo "Hello World!";
    }
  }

  // Call static method
  greeting::welcome();
  ?>
  ```
  ```

- A class can have both static and non-static methods. A static method can be accessed from a method in the same class using the self keyword and double colon (::).
-
```
<?php
class greeting {
  public static function welcome() {
    echo "Hello World!";
  }
  public function __construct() {
    self::welcome();
  }
}
new greeting();
?>
```

- To call a static method from a child class, use the parent keyword inside the child class. Here, the static method can be public or protected.
- <?php
  class domain {
    protected static function getWebsiteName() {
      return “depaul.edu.in";
    }
  }
  class domainDe extends domain {
    public $websiteName;
    public function __construct() {
      $this->websiteName = parent::getWebsiteName();
    }
  }
  $domainDe = new domainDe;
  echo $domainDe -> websiteName;
  ?>

- Static properties can be called directly - without creating an instance of a class.
- Static properties are declared with the static keyword.
- ```
  <?php
  class ClassName {
    public static $staticProp = “Hello";
  }
  ?>
  ```
- ClassName::staticProp;
- ```
  <?php
  class pi {
    public static $value = 3.14159;
  }
  // Get static property
  echo pi::$value;
  ?>
  ```

- A class can have both static and non-static properties. A static property can be accessed from a method in the same class using the self keyword and double colon (::).
- <?php
  class pi {
    public static $value=3.14159;
    public function staticValue() {
      return self::$value;
    }
  }

  $pi = new pi();
  echo $pi->staticValue();
  ?>

- To call a static property from a child class, use the parent keyword inside the child class.
-

```
<?php
class pi {
  public static $value=3.14159;
}
class x extends pi {
  public function xStatic() {
    return parent::$value;
  }
}
// Get value of static property directly via child class
echo x::$value;

// or get value of static property via xStatic() method
$x = new x();
echo $x->xStatic();
?>
```