- **OOP stands for Object-Oriented Programming.**

- **Procedural (اجرائيات) programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.**

- **Object-oriented programming has several advantages over procedural programming:**

  - OOP is faster and easier to execute
  - OOP provides a clear structure for the programs
  - OOP helps to keep the PHP code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
  - OOP makes it possible to create full reusable applications with less code and shorter development time

- **PHP - What are Classes and Objects?**
  - Classes and objects are the two main aspects of object-oriented programming.
  - Look at the following illustration to see the difference between class and objects:

class	**objects**
Fruit	Apple Banana Mango
class	**objects**
Car	Volvo Audi Toyota

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the properties and behaviors from the class, but each object will have different values for the properties.

- **Define a Class**

  - A class is defined by using the class keyword, followed by the name of the class and a pair of curly braces {}. All its properties and methods go inside the braces:

```
<?php
   class Fruit {
      // code goes here...
   }
?>
```

➢ Define Objects

- Classes are nothing without objects! We can create multiple objects from a class. Each object has all the properties and methods defined in the class, but they will have different property values.
- Objects of a class is created using the new keyword.
- In the example below, $apple and $banana are instances of the class Fruit:

```
<?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();
?>
```

- The **$this** keyword refers to the current object, and is only available inside methods.

➢ PHP - instanceof

- You can use the instanceof keyword to check if an object belongs to a specific class:

```
<?php
$apple = new Fruit();
var_dump($apple instanceof Fruit);
```

➢ PHP - The __construct Function

- A constructor allows you to initialize an object's properties upon creation of the object.
- If you create a __construct() function, PHP will automatically call this function when you create an object from a class.
- Notice that the construct function starts with two underscores (__)!
- We see in the example below, that using a constructor saves us from calling the set_name() method which reduces the amount of code:

```php
<?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();
?>
```

➢ PHP - The __destruct Function

- A destructor is called when the object is destructed or the script is stopped or exited.
- If you create a __destruct() function, PHP will automatically call this function at the end of the script.
- Notice that the destruct function starts with two underscores (__)!

```php
<?php
class Fruit {
  public $name;
  public $color;
  function __destruct() {
    echo "The fruit is {$this->name}.";
  }
}
$apple = new Fruit("Apple");
?>
```

➢ PHP - Access Modifiers

- 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
- In the following example we have added three different access modifiers to three properties (name, color, and weight). Here, if you try to set the name property it will work fine (because the name property is public, and can be accessed from everywhere). However, if you try to set the color or weight property it will result in a fatal error (because the color and weight property are protected and private):

```
<?php
class Fruit {
  public $name;
  protected $color;
  private $weight;
}
$mango = new Fruit();
$mango->name = 'Mango'; // OK
$mango->color = 'Yellow'; // ERROR
$mango->weight = '300'; // ERROR
?>
```

- In the next example we have added access modifiers to two functions. Here, if you try to call the set_color() or the set_weight() function it will result in a fatal error (because the two functions are considered protected and private), even if all the properties are public:

```
<?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
?>
```

➢ PHP OOP – Inheritance

- 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}.";
  }
}
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 - Inheritance and the Protected Access Modifier

- we learned that protected properties or methods can be accessed within the class and by classes derived from that class.

```
<?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? ";
  }
}
$strawberry = new Strawberry("Strawberry", "red");  // OK. __construct() is public
$strawberry->message(); // OK. message() is public
$strawberry->intro(); // ERROR. intro() is protected
?>
```

  - In the example above we see that if we try to call a protected method (intro()) from outside the class, we will receive an error. public methods will work fine!

```
<?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
?>
```

- In the example above we see that all works fine! It is because we call the protected method (intro()) from inside the derived class.

➢ PHP - Overriding Inherited Methods

- Inherited methods can be overridden by redefining the methods (use the same name) in the child class.

- Look at the example below. The __construct() and intro() methods in the child class (Strawberry) will override the __construct() and intro() methods in the parent class (Fruit):

```
<?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(); ?>
```

➢ PHP - The final Keyword

- The final keyword can be used to prevent class inheritance or to prevent method overriding.

- The following example shows how to prevent class inheritance:

```
<?php
final class Fruit {
  // some code
}

// will result in error
class Strawberry extends Fruit {
  // some code
}
?>
```

- The following example shows how to prevent method overriding:

```
<?php
class Fruit {
  final public function intro() {
    // some code
  }
}
class Strawberry extends Fruit {
  // will result in error
  public function intro() {
    // some code
  }
}
?>
```

➢ PHP - Class Constants

- 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.
- We can access a constant from outside the class by using the class name followed by the scope resolution operator (::) followed by the constant name, like here:

```
<?php
class Goodbye {
  const LEAVING_MESSAGE = "Thank you for visiting W3Schools.com!";
}
echo Goodbye::LEAVING_MESSAGE;
?>
```

➢ PHP - What are Abstract Classes and Methods?

- 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;
}
?>
```

When inheriting from an abstract class, the child class method must be defined with the same name, and the same or a less restricted access modifier. So, if the abstract method is defined as protected, the child class method must be defined as either protected or public, but not private. Also, the type and number of required arguments must be the same. However, the child classes may have optional arguments in addition.

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() : string;
}
// Child classes
class Audi extends Car {
  public function intro() : string {
    return "Choose German quality! I'm an $this->name!";
  }
}
class Volvo extends Car {
  public function intro() : string {
    return "Proud to be Swedish! I'm a $this->name!";
  }
}
class Citroen extends Car {
  public function intro() : string {
    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();
?>
```

- Example Explained

  The Audi, Volvo, and Citroen classes are inherited from the Car class. This means that the Audi, Volvo, and Citroen classes can use the public $name property as well as the public __construct() method from the Car class because of inheritance.

  But, intro() is an abstract method that should be defined in all the child classes and they should return a string.

- **More Abstract Class Examples**

```php
<?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");
?>
```

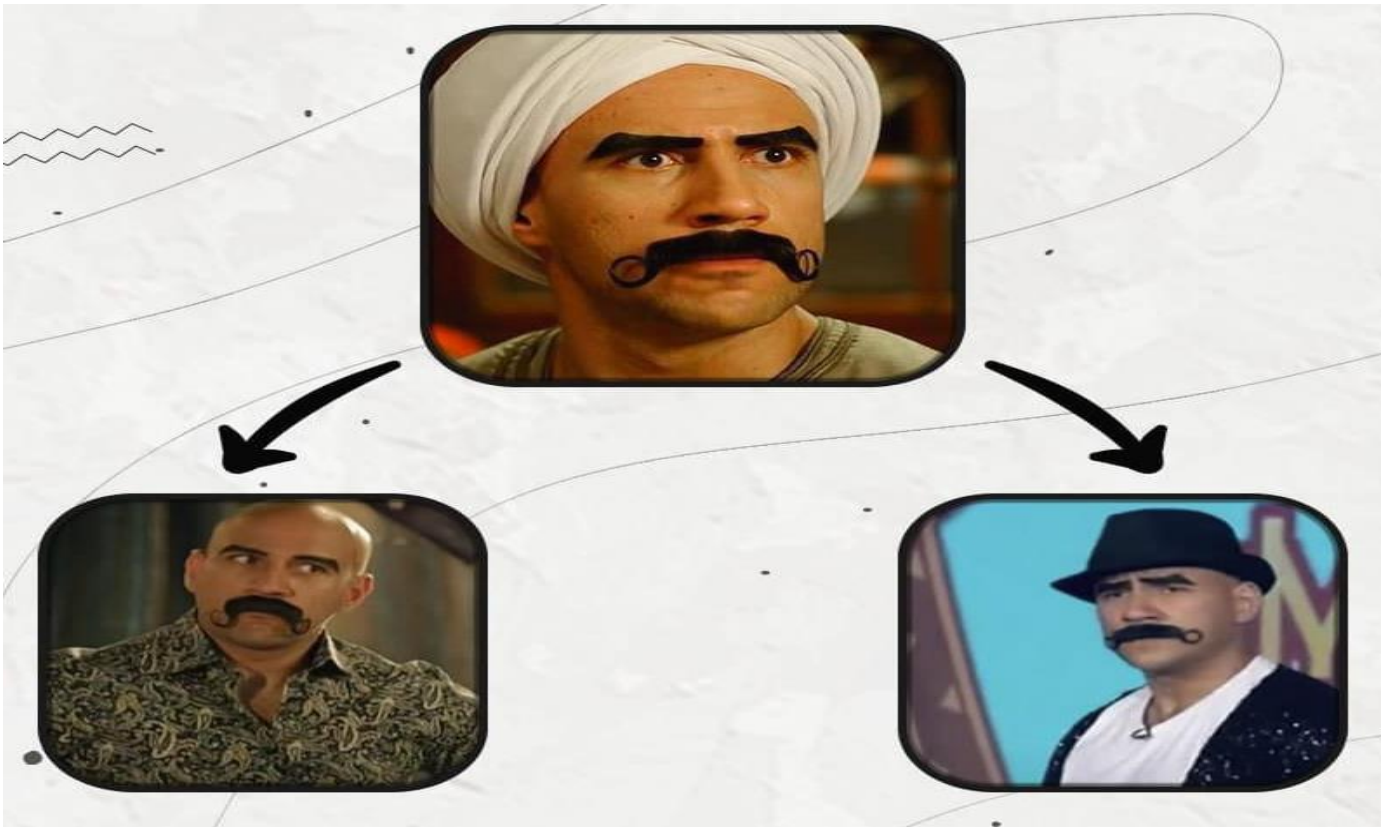
- **Let's look at another example where the abstract method has an argument, and the child class has two optional arguments that are not defined in the parent's abstract method:**

```php
<?php
abstract class ParentClass {
  // Abstract method with an argument
  abstract protected function prefixName($name);
}
class ChildClass extends ParentClass {
  // The child class may define optional arguments that are not in the parent's
abstract method
  public function prefixName($name, $separator = ".", $greet = "Dear") {
    if ($name == "John Doe") {
      $prefix = "Mr";
    } elseif ($name == "Jane Doe") {
      $prefix = "Mrs";
    } else {
      $prefix = "";
    }
```

```
    return "{$greet} {$prefix}{$separator} {$name}";
  }
}

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

- **PHP - What are Interfaces?**



Interfaces allow you to specify what methods a class should implement.

Interfaces make it easy to use a variety of different classes in the same way. When one or more classes use the same interface, it is referred to as "polymorphism".

Interfaces are declared with the interface keyword:

```
<?php
interface InterfaceName {
  public function someMethod1();
  public function someMethod2($name, $color);
  public function someMethod3() : string;
}
?>
```

➢ PHP - Interfaces vs. Abstract Classes

Interface are similar to abstract classes. The difference between interfaces and abstract classes are:

- Interfaces cannot have properties, while abstract classes can
- All interface methods must be public, while abstract class methods is public or protected
- All methods in an interface are abstract, so they cannot be implemented in code and the abstract keyword is not necessary
- Classes can implement an interface while inheriting from another class at the same time

➢ PHP - Using Interfaces

To implement an interface, a class must use the implements keyword.

A class that implements an interface must implement **all** of the interface's methods.

```
<?php
interface Animal {
  public function makeSound();
}

class Cat implements Animal {
  public function makeSound() {
    echo "Meow";
  }
}
$animal = new Cat();
$animal->makeSound();
?>
```

From the example above, let's say that we would like to write software which manages a group of animals. There are actions that all of the animals can do, but each animal does it in its own way.

Using interfaces, we can write some code which can work for all of the animals even if each animal behaves differently:

```
<?php
// Interface definition
interface Animal {
  public function makeSound();
}
// Class definitions
class Cat implements Animal {
  public function makeSound() {
```

```
    echo " Meow ";
  }
}

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

class Mouse implements Animal {
  public function makeSound() {
    echo " Squeak ";
  }
}

// Create a list of animals
$cat = new Cat();
$dog = new Dog();
$mouse = new Mouse();
$animals = array($cat, $dog, $mouse);

// Tell the animals to make a sound
foreach($animals as $animal) {
  $animal->makeSound();
}
?>
```

- **Example Explained**

  Cat, Dog and Mouse are all classes that implement the Animal interface, which means that all of them are able to make a sound using the `makeSound()` method. Because of this, we can loop through all of the animals and tell them to make a sound even if we don't know what type of animal each one is.

  Since the interface does not tell the classes how to implement the method, each animal can make a sound in its own way.