UNIT 4:PHP WITH OOP AND PHP SESSION HANDLING

❖ PHP - What is OOP?

From PHP5, you can also write PHP code in an object-oriented style.

Object-Oriented programming is faster and easier to execute.

OOP stands for Object-Oriented Programming.

Procedural 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

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

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: Fruit

Objects: Apple, Banana, Mango

Another example:

Class: Car

Objects: 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.

❖ PHP OOP - Classes and Objects

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

OOP Case

Let's assume we have a class named Fruit. A Fruit can have properties like name, color, weight, etc. We can define variables like $name, $color, and $weight to hold the values of these properties.

When the individual objects (apple, banana, etc.) 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:

Syntax

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

Below we declare a class named Fruit consisting of two properties ($name and $color) and two methods set_name() and get_name() for setting and getting the $name property:

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

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

Note: In a class, variables are called properties and functions are called methods!

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 are created using the new keyword.

In the example below, $apple and $banana are instances of the class Fruit:

Example

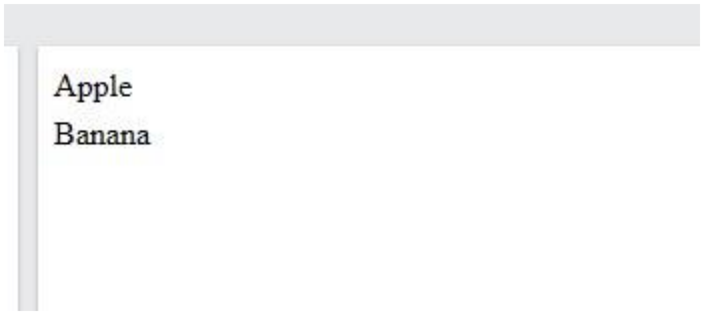
```php
<?php
class Fruit {
  // Properties
  public $name;
  public $color;

  // 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();
?>
```

Output



```
Apple
Banana
```

In the example below, we add two more methods to class Fruit, for setting and getting the $color property:

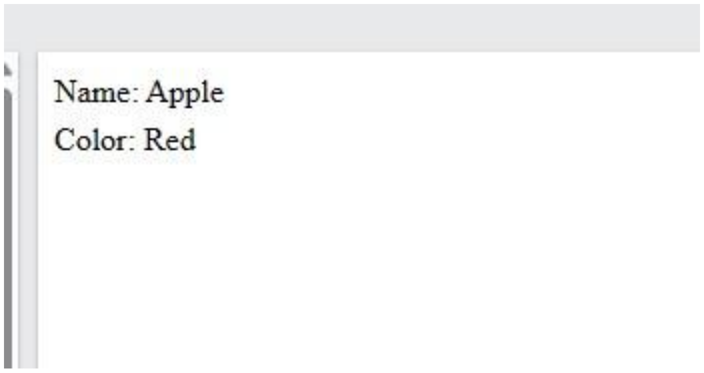
Example

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

Output



```
Name: Apple
Color: Red
```

PHP - The $this Keyword

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

Look at the following example:

Example

```
<?php
class Fruit {
  public $name;
}
$apple = new Fruit();
?>
```

So, where can we change the value of the $name property? There are two ways:

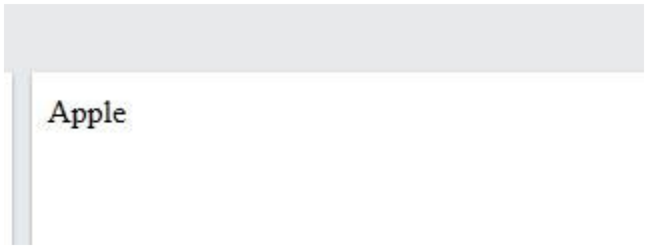
1. Inside the class (by adding a set_name() method and use $this):

Example

```
<?php
class Fruit {
  public $name;
  function set_name($name) {
    $this->name = $name;
  }
}
$apple = new Fruit();
$apple->set_name("Apple");

echo $apple->name;
?>
```

Output

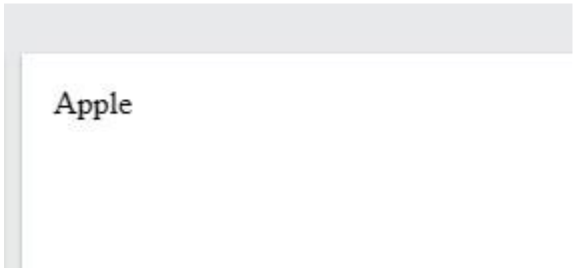

```
Apple
```

2. Outside the class (by directly changing the property value):

Example
```php
<?php
class Fruit {
  public $name;
}
$apple = new Fruit();
$apple->name = "Apple";

echo $apple->name;
?>
```

Output

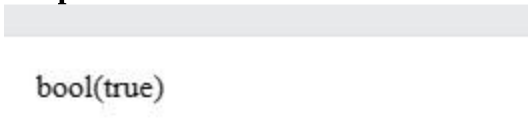

```
Apple
```

PHP - instanceof

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

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

Output



```
bool(true)
```

❖ **PHP OOP - Constructor**

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:

Example

```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();
?>
```

Another example:

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

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

$apple = new Fruit("Apple", "red");
echo $apple->get_name();
echo "<br>";
echo $apple->get_color();
?>
```

❖ <u>**PHP OOP - Destructor**</u>

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 (__)!

The example below has a __construct() function that is automatically called when you create an object from a class, and a __destruct() function that is automatically called at the end of the script:

Example

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

Another example:

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

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

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

❖ **PHP OOP - Access Modifiers**

PHP - Access Modifiers

Properties and methods can have access modifiers which control where they can be accessed.

There are three access modifiers:

1. **public** - the property or method can be accessed from everywhere. This is default
2. **protected** - the property or method can be accessed within the class and by classes derived from that class
3. **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):

Example

```php
<?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:

Example

```php
<?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**

PHP - What is 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.

Let's look at an example:

Example

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

Example Explained

The Strawberry class is inherited from the Fruit class.

This means that the Strawberry class can use the public $name and $color properties as well as the public __construct() and intro() methods from the Fruit class because of inheritance.

The Strawberry class also has its own method: message().

PHP - Inheritance and the Protected Access Modifier

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

Example
```
<?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
?>
```

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!

Let's look at another example:

Example
```
<?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):

Example

```
<?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:

Example

```
<?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:

Example

```
<?php
class Fruit {
  final public function intro() {
    // some code
  }
}

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

❖ PHP OOP - Class Constants

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:

Example

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

echo Goodbye::LEAVING_MESSAGE;
?>
```

Or, we can access a constant from inside the class by using the self keyword followed by the scope resolution operator (::) followed by the constant name, like here:

Example

```
<?php
class Goodbye {
  const LEAVING_MESSAGE = "Thank you for visiting W3Schools.com!";
  public function byebye() {
    echo self::LEAVING_MESSAGE;
  }
}

$goodbye = new Goodbye();
$goodbye->byebye();
?>
```

❖ PHP OOP - Abstract Classes

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:

Syntax

```
<?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

Let's look at an example:

Example

```
<?php
// Parent class
abstract class Car {
  public $name;
  public function __construct($name) {
    $this->name = $name;
```

```php
  }
  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.

❖ PHP - More Abstract Class Examples

Let's look at another example where the abstract method has an argument:

Example

```
<?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:

Example

```
<?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 OOP - Interfaces**

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:

Syntax

```
<?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.

Example

```php
<?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:

Example

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

// Class definitions
class Cat implements Animal {
  public function makeSound() {
    echo " Meow ";
  }
}
```

```php
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.

- ❖ **PHP OOP - Traits**

PHP - What are Traits?

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:

Syntax

```
<?php
trait TraitName {
  // some code...
}
?>
```

To use a trait in a class, use the use keyword:

Syntax

```
<?php
class MyClass {
  use TraitName;
}
?>
```

Let's look at an example:

Example

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

class Welcome {
  use message1;
}

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

Example Explained

Here, we declare one trait: message1. Then, we create a class: Welcome. The class uses the trait, and all the methods in the trait will be available in the class.

If other classes need to use the msg1() function, simply use the message1 trait in those classes. This reduces code duplication, because there is no need to redeclare the same method over and over again.

❖ PHP - Using Multiple Traits

Let's look at another example:

Example

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

Example Explained

Here, we declare two traits: message1 and message2. Then, we create two classes: Welcome and Welcome2. The first class (Welcome) uses the message1 trait, and the second class (Welcome2) uses both message1 and message2 traits (multiple traits are separated by comma).

❖ PHP OOP - Static Methods

PHP - Static Methods

Static methods can be called directly - without creating an instance of the class first.

Static methods are declared with the static keyword:

Syntax

```
<?php
class ClassName {
  public static function staticMethod() {
    echo "Hello World!";
  }
}
?>
```

To access a static method use the class name, double colon (::), and the method name:

Syntax

```
ClassName::staticMethod();
```

Let's look at an example:

Example

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

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

Example Explained

Here, we declare a static method: welcome(). Then, we call the static method by using the class name, double colon (::), and the method name (without creating an instance of the class first).

❖ PHP - More on Static Methods

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 (::):

Example

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

  public function __construct() {
    self::welcome();
  }
}

new greeting();
?>
```

Static methods can also be called from methods in other classes. To do this, the static method should be public:

Example

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

class B {
  public function message() {
    A::welcome();
  }
}

$obj = new B();
echo $obj -> message();
?>
```

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.

Example

```php
<?php
class domain {
  protected static function getWebsiteName() {
    return "W3Schools.com";
  }
}
```

```
class domainW3 extends domain {
  public $websiteName;
  public function __construct() {
    $this->websiteName = parent::getWebsiteName();
  }
}

$domainW3 = new domainW3;
echo $domainW3 -> websiteName;
?>
```

❖ PHP OOP - Static Properties

PHP - Static Properties

Static properties can be called directly - without creating an instance of a class.

Static properties are declared with the static keyword:

Syntax

```
<?php
class ClassName {
  public static $staticProp = "W3Schools";
}
?>
```

To access a static property use the class name, double colon (::), and the property name:

Syntax

```
ClassName::$staticProp;
```

Let's look at an example:

Example

```
<?php
class pi {
  public static $value = 3.14159;
}

// Get static property
echo pi::$value;
?>
```

Example Explained

Here, we declare a static property: $value. Then, we echo the value of the static property by using the class name, double colon (::), and the property name (without creating a class first).

❖ PHP - More on Static Properties

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 (::):

Example
```php
<?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:

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

❖ PHP Namespaces

PHP Namespaces

Namespaces are qualifiers that solve two different problems:

1. They allow for better organization by grouping classes that work together to perform a task

2. They allow the same name to be used for more than one class

For example, you may have a set of classes which describe an HTML table, such as Table, Row and Cell while also having another set of classes to describe furniture, such as Table, Chair and Bed. Namespaces can be used to organize the classes into two different groups while also preventing the two classes Table and Table from being mixed up.

Declaring a Namespace

Namespaces are declared at the beginning of a file using the namespace keyword:

Syntax

Declare a namespace called Html:

```
<?php
namespace Html;
?>
```

Note: A namespace declaration must be the first thing in the PHP file. The following code would be invalid:

```
<?php
echo "Hello World!";
namespace Html;
...
?>
```

Constants, classes and functions declared in this file will belong to the Html namespace:

Example

Create a Table class in the Html namespace:

```
<?php
namespace Html;
class Table {
  public $title = "";
  public $numRows = 0;
  public function message() {
    echo "<p>Table '{$this->title}' has {$this->numRows} rows.</p>";
  }
}
$table = new Table();
$table->title = "My table";
$table->numRows = 5;
?>

<!DOCTYPE html>
<html>
<body>

<?php
$table->message();
?>

</body>
</html>
```

For further organization, it is possible to have nested namespaces:

Syntax

Declare a namespace called Html inside a namespace called Code:

```
<?php
namespace Code\Html;
?>
```

❖ **Using Namespaces**

Any code that follows a namespace declaration is operating inside the namespace, so classes that belong to the namespace can be instantiated without any qualifiers. To access classes from outside a namespace, the class needs to have the namespace attached to it.

Example

Use classes from the Html namespace:

```
<?php
$table = new Html\Table()
```

```php
$row = new Html\Row();
?>
```

When many classes from the same namespace are being used at the same time, it is easier to use the namespace keyword:

Example

Use classes from the Html namespace without the need for the Html\qualifier:

```php
<?php
namespace Html;
$table = new Table();
$row = new Row();
?>
```

- ❖ **Namespace Alias**

It can be useful to give a namespace or class an alias to make it easier to write. This is done with the use keyword:

Example

Give a namespace an alias:

```php
<?php
use Html as H;
$table = new H\Table();
?>
```

Example

Give a class an alias:

```php
<?php
use Html\Table as T;
$table = new T();
?>
```

- ❖ **PHP Iterables**

PHP - What is an Iterable?

An iterable is any value which can be looped through with a foreach() loop.

The iterable pseudo-type was introduced in PHP 7.1, and it can be used as a data type for function arguments and function return values.

PHP - Using Iterables

The iterable keyword can be used as a data type of a function argument or as the return type of a function:

Example

Use an iterable function argument:

```
<?php
function printIterable(iterable $myIterable) {
  foreach($myIterable as $item) {
    echo $item;
  }
}

$arr = ["a", "b", "c"];
printIterable($arr);
?>
```

Example

Return an iterable:

```
<?php
function getIterable():iterable {
  return ["a", "b", "c"];
}

$myIterable = getIterable();
foreach($myIterable as $item) {
  echo $item;
}
?>
```

PHP - Creating Iterables

Arrays

All arrays are iterables, so any array can be used as an argument of a function that requires an iterable.

Iterators

Any object that implements the Iterator interface can be used as an argument of a function that requires an iterable.

An iterator contains a list of items and provides methods to loop through them. It keeps a pointer to one of the elements in the list. Each item in the list should have a key which can be used to find the item.

An iterator must have these methods:

- **current()** - Returns the element that the pointer is currently pointing to. It can be any data type
- **key()** Returns the key associated with the current element in the list. It can only be an integer, float, boolean or string
- **next()** Moves the pointer to the next element in the list
- **rewind()** Moves the pointer to the first element in the list
- **valid()** If the internal pointer is not pointing to any element (for example, if next() was called at the end of the list), this should return false. It returns true in any other case

Example

Implement the Iterator interface and use it as an iterable:

```php
<?php
// Create an Iterator
class MyIterator implements Iterator {
  private $items = [];
  private $pointer = 0;

  public function __construct($items) {
    // array_values() makes sure that the keys are numbers
    $this->items = array_values($items);
  }

  public function current() {
    return $this->items[$this->pointer];
  }

  public function key() {
    return $this->pointer;
  }
  public function next() {
    $this->pointer++;
  }

  public function rewind() {
    $this->pointer = 0;
  }

  public function valid() {
    // count() indicates how many items are in the list
    return $this->pointer < count($this->items);
  }
}

// A function that uses iterables
function printIterable(iterable $myIterable) {
  foreach($myIterable as $item) {
    echo $item;
  }
}

// Use the iterator as an iterable
$iterator = new MyIterator(["a", "b", "c"]);
printIterable($iterator);
?>
```

❖ <u>**PHP Session Handling**</u>

An alternative way to make data accessible across the various pages of an entire website is to use a PHP Session.

A session creates a file in a temporary directory on the server where registered session variables and their values are stored. This data will be available to all pages on the site during that visit.

The location of the temporary file is determined by a setting in the php.ini file called session.save_path. Before using any session variable make sure you have setup this path.

When a session is started following things happen −

- PHP first creates a unique identifier for that particular session which is a random string of 32 hexadecimal numbers such as 3c7foj34c3jj973hjkop2fc937e3443.

- A cookie called PHPSESSID is automatically sent to the user's computer to store unique session identification string.

- A file is automatically created on the server in the designated temporary directory and bears the name of the unique identifier prefixed by sess_ ie sess_3c7foj34c3jj973hjkop2fc937e3443.

When a PHP script wants to retrieve the value from a session variable, PHP automatically gets the unique session identifier string from the PHPSESSID cookie and then looks in its temporary directory for the file bearing that name and a validation can be done by comparing both values.

A session ends when the user loses the browser or after leaving the site, the server will terminate the session after a predetermined period of time, commonly 30 minutes duration.

PHP session_start() function

A PHP session is easily started by making a call to the session_start() function.This function first checks if a session is already started and if none is started then it starts one. It is recommended to put the call to session_start() at the beginning of the page.

Session variables are stored in associative array called $_SESSION[]. These variables can be accessed during lifetime of a session.

The following example starts a session then register a variable called counter that is incremented each time the page is visited during the session.

Make use of isset() function to check if session variable is already set or not.

Put this code in a test.php file and load this file many times to see the result −

```
<?php
   session_start();
   
   if( isset( $_SESSION['counter'] ) ) {
      $_SESSION['counter'] += 1;
   }else {
      $_SESSION['counter'] = 1;
   }
	
   $msg = "You have visited this page ".  $_SESSION['counter'];
   $msg .= "in this session.";
?>

<html>
   
   <head>
      <title>Setting up a PHP session</title>
   </head>
   <body>
      <?php  echo ( $msg ); ?>
   </body>

</html>
```

It will produce the following result −

You have visited this page 1in this session.

PHP session_start() function is used to start the session. It starts a new or resumes existing session. It returns existing session if session is created already. If session is not available, it creates and returns new session.

Syntax

```
bool session_start ( void )
```

Example

```
session_start();
```

PHP $_SESSION

PHP $_SESSION is an associative array that contains all session variables. It is used to set and get session variable values.

Example: Store information

```
$_SESSION["user"] = "Sachin";
```

Example: Get information

```
echo $_SESSION["user"];
```

PHP Session Example

File: session1.php

```
<?php
session_start();
?>
<html>
<body>
<?php
$_SESSION["user"] = "Sachin";
echo "Session information are set successfully.<br/>";
?>
<a href="session2.php">Visit next page</a>
```

```
</body>
</html>
```

File: session2.php

```
<?php
session_start();
?>
<html>
<body>
<?php
echo "User is: ".$_SESSION["user"];
?>
</body>
</html>
```

PHP Session Counter Example

File: sessioncounter.php

```
<?php
  session_start();

  if (!isset($_SESSION['counter'])) {
    $_SESSION['counter'] = 1;
  } else {
    $_SESSION['counter']++;
  }
  echo ("Page Views: ".$_SESSION['counter']);
?>
```

PHP Destroying Session

PHP session_destroy() function is used to destroy all session variables completely.

File: session3.php

```
<?php
session_start();
```

```
        session_destroy();
        ?>
```

Starting a PHP Session

Destroying a PHP Session

A PHP session can be destroyed by session_destroy() function. This function does not need any argument and a single call can destroy all the session variables. If you want to destroy a single session variable then you can use unset() function to unset a session variable.

Here is the example to unset a single variable −

```
<?php
   unset($_SESSION['counter']);
?>
```

Here is the call which will destroy all the session variables −

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

Turning on Auto Session

You don't need to call start_session() function to start a session when a user visits your site if you can set session.auto_start variable to 1 in php.ini file.

Sessions without cookies

There may be a case when a user does not allow to store cookies on their machine. So there is another method to send session ID to the browser.

Alternatively, you can use the constant SID which is defined if the session started. If the client did not send an appropriate session cookie, it has the form session_name=session_id. Otherwise, it expands to an empty string. Thus, you can embed it unconditionally into URLs.

The following example demonstrates how to register a variable, and how to link correctly to another page using SID.

```
<?php
   session_start();

   if (isset($_SESSION['counter'])) {
      $_SESSION['counter'] = 1;
   }else {
      $_SESSION['counter']++;
   }

   $msg = "You have visited this page ".  $_SESSION['counter'];
   $msg .= "in this session.";

   echo ( $msg );
?>

<p>
   To continue  click following link <br />

   <a  href = "nextpage.php?<?php echo htmlspecialchars(SID); ?>">
</p>
```

It will produce the following result −

You have visited this page 1in this session.

To continue click following link

The htmlspecialchars() may be used when printing the SID in order to prevent XSS related attacks.

Example1:

A simple example to understand working with sessions in PHP. In this, there are two PHP programs, first program (**FirstPage.php**) is to create session variables and store information. And the second program (**SecondPage.php**) is to get the information from the session variable.

Example: "FirstPage.php"

```
<?php
// to start the session
session_start();
?>
<html>
<body>
<?php
// To create session variable called "name"
$_SESSION["name"] = "Study Glance";
?>
<a href="SecondPage.php">Click Here to Visit Next Page</a>
</body>
</html>
```

Example: "SecondPage.php"

```
<?php
// to start the session
session_start();
?>
<html>
<body>
<?php
// to get the infromation from session variable
echo "Website Name is: ".$_SESSION["name"];
?>
</body>
</html>
```

Example2:

In this example, reading input from a text field and storing it in a session variable after which another page reads the value of the session variable.

Example: "sessionPage1.php"

```
<?php
// To start the session
session_start();
?>
<html>
<head>
<title>Session example</title>
</head>
<body>
<form name="form1" method="post">
Name : <input name="uname" type="text">
<br/><br/>
<input type="submit" name="Submit" value="SUBMIT">
</form>
<?php
if(isset($_POST['Submit']))
{
$_SESSION["name"] = $_POST["uname"];
header('Location: sessionPage2.php');
}
?>
</body>
</html>
```

Example: "sessionPage2.php"

```
<?php
// To start the session
session_start();
?>
<html>
<head>
<title>Welcome </title>
</head>
<body>
<br/>
<?php
echo "Welcome : ".$_SESSION['name'];
?>
</body>
</html>
```