PHP

Object-Oriented Programming

Mohammed Safadi
PHP Expert and Consultant

Classes and Objects

Class Definition

```php
<?php
class SimpleClass
{
    // property declaration
    public $var = 'a default value';

    // method declaration
    public function displayVar() {
        echo $this->var;
    }
}
```

The pseudo-variable `$this` is available when a method is called from within an object context. `$this` is the value of the calling object.

Class Instance (Object)

```php
<?php
$instance = new SimpleClass();

// () after the class name may be omitted, if there are no
arguments to be passed to the class's constructor
$instance2 = new SimpleClass;

// This can also be done with a variable:
$className = 'SimpleClass';
$instance3 = new $className(); // new SimpleClass()
?>
```

Object Assignment

```php
<?php

$instance = new SimpleClass();

$assigned   =  $instance; // Copies of the same identifier
$reference  =& $instance; // References

$instance->var = '$assigned will have this value';

$instance = null; // $instance and $reference become null
```

Object Cloning

An object copy is created by using the clone keyword

```php
<?php

$instance  = new SimpleClass();

$copy      = clone $instance;

$copy->var = '$instance will not have this value';
```

Tips

To obtain the fully qualified name of a class ClassName use **`ClassName::class`**.
As of **PHP 8.0.0**, properties and methods may also be accessed with the "nullsafe" operator instead: **`?->`**

```php
<?php

echo SimpleClass::class; // Output: SimpleClass

$result = $repository?->name;

// Is equivalent to the following code block:
if (is_null($repository)) {
    $result = null;
} else {
    $user = $repository->name;
}
?>
```

Properties

- Class member variables are called **properties**.
- Within class methods non-static properties may be accessed by using **->** (Object Operator): **`$this->property`** (where property is the name of the property).
- Static properties are accessed by using the **`::`** (Double Colon): **`self::$property`**.
- As of PHP 7.4.0, property definitions can include Type declarations
- As of PHP 8.1.0, a property can be declared with the **readonly** modifier, which prevents modification of the property after initialization.
- The readonly modifier can only be applied to typed properties.

readonly Properties

```php
<?php

class Test {
    public readonly string $prop;

    public function __construct(string $prop) {
        // Legal initialization.
        $this->prop = $prop;
    }
}

$test = new Test("foobar");
// Legal read.
echo $test->prop; // foobar

$test->prop = "foobar";
// Error: Cannot modify readonly property Test::$prop
```

Class Constants

```php
<?php
class MyClass
{
    const CONSTANT = 'constant value';

    function showConstant() {
        echo  self::CONSTANT . "\n";
    }
}

echo MyClass::CONSTANT . "\n";

$classname = "MyClass";
echo $classname::CONSTANT . "\n";

$class = new MyClass();
$class->showConstant();

echo $class::CONSTANT."\n";
```

Constructors and Destructors

```
class Point {
    protected int $x;
    protected int $y;

    public function __construct(int $x, int $y = 0) {
        $this->x = $x;
        $this->y = $y;
    }

    function __destruct() {
        print "Destroying " . __CLASS__ . "\n";
    }
}
// Pass both parameters.
$p1 = new Point(4, 5);
// Pass only the required parameter. $y will take its default value of 0.
$p2 = new Point(4);
```

Visibility

- The visibility of a property, a method or (as of PHP 7.1.0) a constant can be defined by prefixing the declaration with the keywords **public, protected or private**.
- Class members declared public can be accessed everywhere.
- Members declared protected can be accessed only within the class itself and by inheriting and parent classes.
- Members declared as private may only be accessed by the class that defines the member.
- Members declared without any explicit visibility keyword are defined as **public**.

Visibility Example

```php
class MyClass {
    protected const MY_PROTECTED = 'protected';

    public $public = 'Public';
    protected $protected = 'Protected';
    private $private = 'Private';

    function printHello() {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj = new MyClass();
echo $obj->public;    // Works
echo $obj->protected; // Fatal Error
echo $obj->private;   // Fatal Error
$obj->printHello();   // Shows Public, Protected and Private
```

Object Inheritance

```php
<?php
class ExtendClass extends SimpleClass
{
    // Redefine the parent method
    function displayVar()
    {
        echo "Extending class\n";
        parent::displayVar();
    }
}

$extended = new ExtendClass();
$extended->displayVar();
```

When overriding a method, its signature must be compatible with the parent method.

Renaming a method's parameter in a child class is not a signature incompatibility. However, this is discouraged as it will result in a runtime Error if named arguments are used.

Final Keyword

```php
<?php
class BaseClass {
   public function test() {
       echo "BaseClass::test() called\n";
   }

   final public function moreTesting() {
       echo "BaseClass::moreTesting() called\n";
   }
}

class ChildClass extends BaseClass {
   public function moreTesting() {
       echo "ChildClass::moreTesting() called\n";
   }
}
// Results in Fatal error: Cannot override final method BaseClass::moreTesting()
```

The final keyword prevents child classes from overriding a method or constant by prefixing the definition with final. If the class itself is being defined final then it cannot be extended.

Scope Resolution Operator (::)

- Allows access to static, constant, and overridden properties or methods of a class.
- When referencing these items from outside the class definition, use the name of the class.

```
class OtherClass extends MyClass
{
    public static $my_static = 'static var';

    public static function doubleColon() {
        echo parent::CONST_VALUE . "\n";
        echo self::$my_static . "\n";
    }
}

$classname = 'OtherClass';
$classname::doubleColon();

OtherClass::doubleColon();
```

Static methods

```php
<?php
class Foo {
    public static function aStaticMethod() {
        // ...
    }
}

Foo::aStaticMethod();
$classname = 'Foo';
$classname::aStaticMethod();
?>
```

Because static methods are callable without an instance of the object created, the pseudo-variable `$this` is not available inside methods declared as static.

Static properties

```php
<?php

class Foo
{
    public static $my_static = 'foo';

    public function staticValue() {
        return self::$my_static;
    }
}

class Bar extends Foo
{
    public function fooStatic() {
        return parent::$my_static;
    }
}
```

```php
echo Foo::$my_static . "\n";

$foo = new Foo();

echo $foo->staticValue() . "\n";

// Undefined "Property" my_static
echo $foo->my_static . "\n";

echo $foo::$my_static . "\n";
```

Abstract Classes

```
abstract class AbstractClass {
    // Force Extending class to define this method
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);

    public function printOut() {
        print $this->getValue() . "\n";
    }
}

class ConcreteClass1 extends AbstractClass {
    protected function getValue() {
        return "ConcreteClass1";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass1";
    }
}
```

Classes defined as abstract cannot be instantiated, and any class that contains at least one abstract method must also be abstract.

Interfaces

```php
interface Shape
{
    public function getPerimeter();
    public function getArea();
}

class Square implements Shape
{
    private $side;

    public function getPerimeter()
    {
        return $this->side * 4;
    }

    public function getArea()
    {
        return $this->side * $this->side;
    }
}
```

All methods declared in an interface must be public; this is the nature of an interface.

It's possible for interfaces to have constants

The class implementing the interface must declare all methods in the interface with a compatible signature.

Classes may implement more than one interface if desired by separating each interface with a comma.

Exercise

- Write a PHP class "Str" that accept a string in its constructor:
  - Write a method "length" that return the length of the string
  - Write a method "equal" that accept another string and return true if the two string is identical.
  - Write a static method that return the string value. Does it work?
  - Write the required code to test your class.

Traits

- Traits are a mechanism **for code reuse** in single inheritance languages such as PHP.
- A Trait is similar to a class, but only intended to group functionality in a fine-grained and consistent way.

```php
<?php

trait HelloWorld {
    public function sayHello() {
        echo 'Hello World!';
    }
}
```

```php
class TheWorldIsNotEnough {
    use HelloWorld;

    public function sayHello() {
        echo 'Hello Universe!';
    }
}

$o = new TheWorldIsNotEnough();
$o->sayHello();
```

Multiple Traits

```php
<?php
trait Hello {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait World {
    public function sayWorld() {
        echo 'World';
    }
}

class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {
        echo '!';
    }
}
```

Traits: Conflict Resolution

```php
trait A {
    public function smallTalk() {
        echo 'a';
    }
    public function bigTalk() {
        echo 'A';
    }
}
```

```php
trait B {
    public function smallTalk() {
        echo 'b';
    }
    public function bigTalk() {
        echo 'B';
    }
}
```

```php
class Talker {
    use A, B {
        B::smallTalk insteadof A;
        A::bigTalk insteadof B;
    }
}
```

```php
class Aliased_Talker {
    use A, B {
        B::smallTalk insteadof A;
        A::bigTalk insteadof B;
        B::bigTalk as talk;
    }
}
```

Anonymous classes

```php
<?php

class SomeClass {}
interface SomeInterface {}
trait SomeTrait {}

$obj = new class(10) extends SomeClass implements SomeInterface {
    private $num;

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

    use SomeTrait;
};
```

Namespaces

Introduction

- In the PHP world, namespaces are designed to solve two problems that authors of libraries and applications encounter when creating reusable code elements such as classes or functions:
  - Name collisions between code you create, and internal PHP classes/functions/constants or third-party classes/functions/constants.
  - Ability to alias (or shorten) Extra_Long_Names designed to alleviate the first problem, improving readability of source code.
- PHP Namespaces provide a way in which to group related classes, interfaces, functions and constants.
- Namespaces are declared using the namespace keyword.
- A file containing a namespace must declare the namespace at the top of the file before any other code

Namespace Syntax Example

```php
<?php
namespace my\name; // see "Defining Namespaces" section

class MyClass {}
function myfunction() {}
const MYCONST = 1;

$a = new MyClass;
$c = new \my\name\MyClass; // see "Global Space" section

$a = strlen('hi'); // see "Using namespaces: fallback to global
                   // function/constant" section

$d = namespace\MYCONST; // see "namespace operator and __NAMESPACE__
                        // constant" section
$d = __NAMESPACE__ . '\MYCONST';
echo constant($d); // see "Namespaces and dynamic language features" section
```

Using namespaces: Aliasing/Importing

```php
namespace foo;

use My\Full\Classname as Another;

// this is the same as use My\Full\NSname as NSname
use My\Full\NSname;
// importing a global class
use ArrayObject;
// importing a function
use function My\Full\functionName;
// aliasing a function
use function My\Full\functionName as func;
// importing a constant
use const My\Full\CONSTANT;

$obj = new namespace\Another; // instantiates object of class foo\Another
$obj = new Another; // instantiates object of class My\Full\Classname
NSname\subns\func(); // calls function My\Full\NSname\subns\func
$a = new ArrayObject(array(1)); // instantiates object of class ArrayObject
// without the "use ArrayObject" we would instantiate an object of class foo\ArrayObject
func(); // calls function My\Full\functionName
echo CONSTANT; // echoes the value of My\Full\CONSTANT
```

Importing and fully qualified names

```php
<?php

use My\Full\Classname as Another, My\Full\NSname;

// instantiates object of class My\Full\Classname
$obj = new Another;

// instantiates object of class Another
$obj = new \Another;

// instantiates object of class My\Full\Classname\thing
$obj = new Another\thing;

// instantiates object of class Another\thing
$obj = new \Another\thing;
```

Autoloading Classes

```php
<?php
/*
Attempts to load the classes MyClass1 and MyClass2 from the files
MyClass1.php and MyClass2.php respectively.
*/
spl_autoload_register(function ($class_name) {
    include $class_name . '.php';
});

$obj  = new MyClass1();
$obj2 = new MyClass2();
```

Any class-like construct may be autoloaded the same way. That includes classes, interfaces, traits, and enumerations.

Exercise

- Create a PHP class "Employee" belongs to "Company" namespace that accept employee's name.
- Create another class "Employee" belongs to the PHP global namespace that accept employee's name.
- Define a trait "Greeting" that contain one method "hello" that print "Hello, {name}" where {name} is the name of the employee. Use this trait inside both "Employee" classes. (Hint: You can use $this inside traits)
- Test your code inside a separate file that belongs to a namespace "Test".
- Use the PHP autoload code below to autoload your classes:

```php
spl_autoload_register(function ($class_name) {

    include __DIR__ . "/classes/{$class_name}.php";
});
```