Classes, Methods, Properties

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!

OOP in PHP: Basic Terms

Class – a 'blueprint' to create objects.

Properties – class data.

Constants – constant class data.

Methods – class logic (action mechanisms).

Object – an instance of a class.

```php
<?php

class SampleClass
{
    public string $publicProperty = '';
    protected bool $protectedProperty = true;
    private int $privateProperty = 0;

    public const PUBLIC_CONST = '';
    protected const PROTECTED_CONST = '';
    private const PRIVATE_CONST = '';

    public function __construct() {}

    public function publicMethod() {}
    protected function rotectedMethod() {}
    private function privateMethod() {}

    public function __destruct() {}
}

$someObject = new SomeClass();
```

Constructor – a special method automatically called upon object creation.

Destructor – a special method automatically called upon object destruction.

Visibility (scope): public, protected, private – in general

public	“Everywhere” (within namespace or any “parent scope”)
protected	Inside a class and its descendants
private	Inside the class itself only

Properties visibility (scope) and other modifiers

public	“Everywhere” (within namespace or any “parent scope”)
protected	Inside a class and its descendants
private	Inside the class itself only
static	A single value for all instances
final	“Non-overridable” class constant
~~abstract~~	A property can not be an abstract one

Properties visibility (scope) and other modifiers

A **public** property is accessible from outside of the class code. Avoid at all costs!

It violates encapsulation principle.

A **protected** property is accessible from the class itself and from its descendants.

A **private** property is accessible from the class itself only.

```php
<?php

class StringsManipulator
{
    public string $currentStringRawRepresentation = '';
    protected string $currentString = '';
    protected string $uniqueId = '';
    private string $typeDependantHash = '';

    public const ENCODIND_DEPENDANT = 0b0001;
    public const ENCODIND_INDEPENDANT = 0b0010;

    public final const STRING_TEXT = 0b0001;
    public final const STRING_BINARY = 0b0010;

    public static int $justACounter = 0;

    public function __construct(string $initialString)
    {
        $this->currentString = $initialString;
        $this->uniqueId = uniqid();
        self::$justACounter++;
    }

}

$stringOne = new StringsManipulator('ABC');
$stringTwo = new StringsManipulator('DEF');

echo $stringOne::$justACounter . "\n"; // 2
echo $stringTwo::$justACounter . "\n"; // 2
```

These constants are overridable in descendants.

These **final** constants are non-overridable in descendants.

This **static** property will have the same value for all objects (instances).

*This **static** property will have the same value for all objects (instances).*

Why do we need class constants?

Class constants allow to avoid ‘magic numbers’, to increase code maintainability, to decrease the probability of a mistake, and so on…

```php
<?php

class HttpConnection
{
    const PENDING_RESPONSE   = 0b0001;
    const RECEIVENING_HEADER = 0b0010;
    const RECEIVENING_DATA   = 0b0100;
    // ...
}

// Nice way to use class constants:
if ($httpConnection->getStatus() == HTTPConnection::RECEIVENING_DATA) {
    // ...
}

// Do NOT do this! This is WRONG!
// This is so-called 'magic number' -- extremely BAD practice!
if ($httpConnection->getStatus() == 4) {
    // ...
}
```

Methods visibility (scope) and other modifiers

public	“Everywhere” (within namespace or any “parent scope”)
protected	Inside a class and its descendants
private	Inside the class itself only
static	Accessible via class name, i.e. without an object
final	Non-overridable (in descendants)
abstract	Has to be overridden in descendants

Methods visibility (scope) and other modifiers

A **public** method is accessible from outside of the class code.

A **protected** method is accessible from the class itself and from its descendants.

A **private** method is accessible from the class itself only.

A **final** method is non-overridable in descendants.

An **abstract** method has to be overridden in descendants.

```php
<?php

class StringsManipulator
{
    public string $currentStringRawRepresentation = '';
    protected string $currentString = '';
    protected string $uniqueId = '';
    private string $typeDependantHash = '';

    public function __construct(string $initialString)
    {
        $this->currentString = $initialString;
        $this->uniqueId = uniqid();
    }

    protected function calculateHash(string $inputString) : string
    {
        return '';
    }

    private function typeDependantHash(string $inputString) : string
    {
        return '';
    }

    final public function getCryptoHash(string $inputString) : string
    {
        return '';
    }

    abstract public function getNationalHash(string $inputString) : string;

}
```

If a class has at least one **abstract** method, the whole class itself becomes an abstract one, i.e. it can not be instantiated.

Why do we need static methods?

Static methods are used in two main cases:

1) In some design patterns (like “singleton”).
2) In case method behavior does not depend on object state.

```php
<?php

class Config
{
    private static $instance;

    protected function __construct() {}
    protected function __clone() {}

    public function __wakeup()
    {
        throw new Exception("Cannot unserialize a singleton.");
    }

    public static function getInstance(): Config
    {
        if (is_null(self::$instance)) {
            self::$instance = new Config;
        }
        return self::$instance;
    }

}

$config = Config::getInstance();
```

```php
<?php

class Math
{
    public static function inc($x, $increment = 1)
    {

        return $x + $increment;

    }
}

echo Math::inc(5); // 6
```

Class modifiers

abstract	Instantiation is prohibited, only descendants may be instantiated
final	Inheritance is prohibited

Why do we need abstract and final classes?

Abstract classes are useful as “the beginning” of a hierarchy.
Final classes are useful to prevent any overrides that may cause problems.

```php
<?php

abstract class DbmsConnection
{
    abstract public function connect() : bool;
}

class MySqlConnection extends DbmsConnection {
    public function connect() : bool {
        return true;
    }
}

class OracleConnection extends DbmsConnection
{
    public function connect() : bool {
        return true;
    }
}
```

```php
<?php

final class ExtremelyStrongEncryption
{
    public function encryptForever() : string
    {
        return '';
    }
}
```

Once again: static, final, abstract – just to remember

Modifier	Applicable to…			Effect
	Class	Method	Property	
final	Yes	Yes	Yes	• A class inheritance is prohibited. • A method override is prohibited. • A constant change is prohibited.
static	No	Yes	Yes	• A method is callable via class name (without an object). • An attribute shares its value across all objects (instances).
abstract	Yes	Yes	No	• A class instantiation is prohibited, only descendants may be instantiated. • A method has no implementation, i.e. has to be implemented in class descendants.

And some more keywords: this, self, static, parent

this	Access to the object scope.
self	Early binding, access to the scope of a class a method was **declared** in.
static	Late binding, access to the scope of a class a method was **called** in.
parent	Access to the parent class scope.

And some more keywords: this

In PHP **$this** literally means “the object we are inside now”.

```php
<?php

class SomeClass
{
    private $property;

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

    private function initSomething()
    {

    }
}

$someObject = new SomeClass(999);
```

This doesn’t work without **$this**.

And some more keywords: self, early binding

Early binding means access to the scope of a class a method was **declared** in.

```php
<?php

class A
{
    public static function whoAmI()
    {
        echo __CLASS__;
    }

    public static function justSomeTest()
    {
        self::whoAmI();
    }
}

class B extends A
{
    public static function whoAmI()
    {
        echo __CLASS__;
    }
}

B::justSomeTest(); // A
```

While the method was called from B class, due to **self** keyword the implementation of the method in A class was really invoked.

And some more keywords: static, late binding

Late binding means access to the scope of a class a method was **called** in.

```php
<?php

class A
{
    public static function whoAmI()
    {
        echo __CLASS__;
    }

    public static function justSomeTest()
    {
        static::whoAmI();
    }
}

class B extends A
{
    public static function whoAmI()
    {
        echo __CLASS__;
    }
}

B::justSomeTest(); // B
```

The inherited implementation of the method in B class was really invoked.

And some more keywords: parent, access to the parent class scope

The **parent** keyword allows access to the parent class scope.

```php
<?php

class A
{
    public function example()
    {
        echo 'A';
    }
}

class B extends A
{
    public function example()
    {
        echo 'B';
        parent::example();
    }
}

$b = new B;
$b->example(); // BA
```

This is how we access parent class scope.

Objects comparison

With **==** instances are considered equal if they have the same attributes and values (values are compared with ==), and are instances of the same class.

With **===** object variables are considered equal if and only if they refer to the same instance of the same class.

Objects comparison

```php
<?php

class SomeClass {
    public string $someValue;
}

$objectOne = new SomeClass();
$objectTwo = new SomeClass();

$objectOne->someValue = 'A';
$objectTwo->someValue = 'A';
$objectThree = $objectTwo;
var_dump($objectOne == $objectTwo);     // true
var_dump($objectOne === $objectTwo);    // false
var_dump($objectTwo === $objectThree); // true

$objectOne->someValue = 'X';
$objectTwo->someValue = 'Y';
var_dump($objectOne == $objectTwo);     // false
```

Classes, Methods, Properties

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!