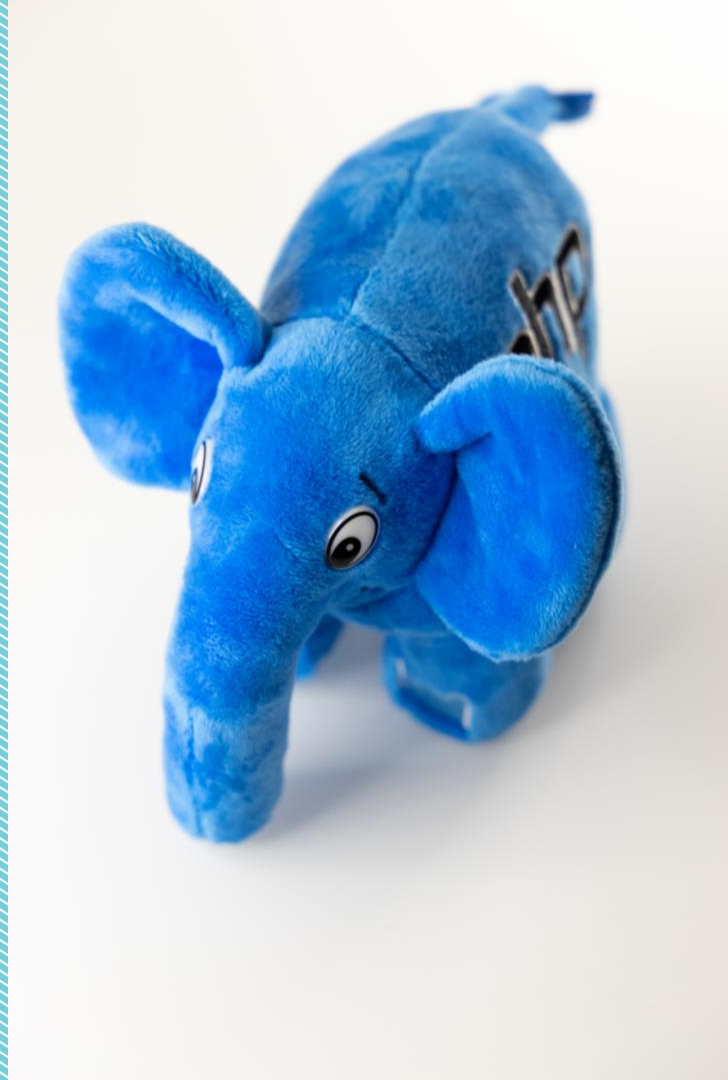
# Magic Methods, Interfaces, Traits, Attributes

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



# Magic methods

Magic methods are special methods which override PHP's default's action when certain actions are performed on an object.

- __construct() and __destruct()
- __call() and __callStatic()
- __get() and __set()
- __isset() and __unset()
- __sleep() and __wakeup()
- __serialize() and __unserialize()
- __toString()
- __invoke()
- __set_state()
- __clone()
- __debugInfo()

Most other programming languages have the same methos. Yet PHP is the one hones enough to call them “magic” methods ☺.

Some of these methods may be useful from the beginning.

And some of these methods may “skip” for now.

# Magic methods: __construct() and __destruct()

**Constructor** is called upon object creation (useful for initialization).
**Destructor** is called upon object destruction (useful for “cleaning”).

```php
<?php

class TempFileManager
{
    private string $fileName;

    public function __construct($userDefinedFileName = '')
    {
        if ($userDefinedFileName != '') {
            $this->fileName = $userDefinedFileName;
        } else {
            $this->fileName = md5(rand());
        }
    }

    public function __destruct()
    {
        if (is_file($this->fileName)) {
            unlink($this->fileName);
        }
    }
}
```

Constructor is called upon object creation (useful for initialization).

Destructor is called upon object destruction (useful for “cleaning”).

# Magic methods: __construct() since PHP 8

Since PHP 8 there is another way to initialize object properties with data received by a constructor:

```php
<?php

// Before PHP 8
class Money
{
    public Currency $currency;
    public int $amount;

    public function __construct(
        Currency $currency,
        int $amount,
    )
    {
        $this->currency = $currency;
        $this->amount = $amount;
    }
}
```

```php
<?php

// Since PHP 8
class NewMoney
{
    public function __construct(
        public Currency $currency,
        public int $amount,
    )
    {
    }
}
```

# Magic methods: __call() and __callStatic()

**__call()** is triggered when invoking inaccessible methods in an object context.
**__callStatic()** does the same but in a static context.

```php
<?php

class SomeClass
{
    public function __call($name, $arguments)
    {
        echo 'Method [' . $name . '] was called with arguments [' .
             implode(', ', $arguments) . "], but it does not exist!\n";
    }

    public static function __callStatic($name, $arguments)
    {
        echo 'Static method [' . $name . '] was called with arguments [' .
            implode(', ', $arguments) . "], but it does not exist!\n";
    }
}

$someObject = new SomeClass;
$someObject->runTest('A', 'B', 'C');
// Method [runTest] was called with arguments [A, B, C], but it does not exist!

SomeClass::runTest(1, 2, 3);
// Static method [runTest] was called with arguments [1, 2, 3], but it does not exist!
```

__call() is triggered when invoking inaccessible methods in an object context.

__callStatic() does the same but in a static context.

# Magic methods: __get() and __set()

**__get()** is triggered on reading data from inaccessible or non-existing property.
**__set()** is triggered on writing data to inaccessible or non-existing property.

```php
<?php

class SomeClass
{
    public function __get($name)
    {
        echo 'Property [' . $name . "] does not exist or is inaccessible!\n";
    }

    public function __set($name, $value)
    {
        echo 'Property [' . $name . "] does not exist or is inaccessible! The value [" .
             $value . "] was not assigned!\n";
    }
}

$someObject = new SomeClass;
$someObject->someProperty;
// Property [someProperty] does not exist or is inaccessible!

$someObject->someProperty = 'someValue';
// Property [someProperty] does not exist or is inaccessible! The value [someValue] was not assigned!
```

__get() is triggered on reading data from inaccessible or non-existing property.

__set() is triggered on writing data to inaccessible or non-existing property.

# Magic methods: __isset() and __unset()

**__isset()** is triggered by calling isset() or empty() on inaccessible or non-existing property.
**__unset()** is triggered when unset() is used on inaccessible or non-existing property.

```php
<?php

class SomeClass
{
    public function __isset($name)
    {
        echo 'Property [' . $name . "] does not exist or is inaccessible!\n";
    }

    public function __unset($name)
    {
        echo 'Property [' . $name . "] does not exist or is inaccessible!\n";
    }
}

$someObject = new SomeClass;
if (isset($someObject->someProperty)) {
    // Do something.
}
// Property [someProperty] does not exist or is inaccessible!

unset ($someObject->someProperty);
// Property [someProperty] does not exist or is inaccessible!
```

__isset() is triggered by calling isset() or empty() on inaccessible or non-existing property.

__unset() is triggered when unset() is used on inaccessible or non-existing property.

# Magic methods: __sleep(), __wakeup(), __serialize(), __unserialize()

**serialize()** checks if the class has a function with the name **__sleep()**. If so, that function is executed prior to any serialization.

**unserialize()** checks for the presence of a function with the name **__wakeup()**. If present, this function can reconstruct any resources that the object may have.

The intended use of **__serialize()** is to define a serialization-friendly arbitrary representation of the object. Elements of the array may correspond to properties of the object but that is not required.

Conversely, **unserialize()** checks for the presence of a function with the name **__unserialize()**. If present, this function will be passed the restored array that was returned from **__serialize()**. It may then restore the properties of the object from that array as appropriate.

# Magic methods: __sleep(), __wakeup(), __serialize(), __unserialize()

```php
<?php

class Connection
{
    protected $link;
    private $connectionString, $username, $password;
    private $lastActionTimeStamp;

    public function __construct($connectionString, $username, $password)
    {
        echo "Constructor called\n";
        $this->connectionString = $connectionString;
        $this->username = $username;
        $this->password = $password;
        $this->connect();
    }

    public function __destruct() {
        echo "Destructor called\n";
        $this->close();
    }

    private function connect()
    {
        echo "connect() called\n";
        $this->lastActionTimeStamp = microtime(true);
    }

    private function close()
    {
        echo "close() called\n";
        $this->lastActionTimeStamp = microtime(true);
    }

    public function __sleep()
    {
        echo "__sleep() called\n";
        return array('connectionString', 'username', 'password');
    }

    public function __wakeup()
    {
        echo "__wakeup() called\n";
        $this->connect();
    }
```

```php
    public function __serialize(): array
    {
        echo "__serialize() called\n";
        return [
            'connectionString' => $this->connectionString,
            'credentials' => ['username' => $this->username,
                              'password' => $this->password],
        ];
    }

    public function __unserialize(array $data): void
    {
        echo "__unserialize() called\n";
        $this->connectionString = $data['connectionString'];
        $this->username = $data['credentials']['username'];
        $this->password = $data['credentials']['password'];
    }

}

$connection = new Connection("mysql:127.0.0.1", "user1", "password1");
$serializedConnection = serialize($connection);
print_r($serializedConnection);
unset($connection);
$connection = unserialize($serializedConnection);
print_r($connection);
```

**__serialize()** defines a serialization-friendly representation of the object.

**__unserialize()** uses data from *__serialize* to restore the object.

**__sleep()** is executed prior to serialization.

**__wakeup()** is executed after unserialization.

If both __serialize() and __sleep() are defined in the same object, only __serialize() will be called.

If both __unserialize() and __wakeup() are defined in the same object, only __unserialize() will be called.

```
/*
Constructor called
connect() called
__serialize() called
O:10:"Connection":2:{s:16:"connectionString";s:15:"mysql:127.0.0.1";s:11:"credentials";a:2:{s:8:
"username";s:5:"user1";s:8:"password";s:9:"password1";}}
Destructor called
close() called
__unserialize() called
Connection Object
(
    [link:protected] =>
    [connectionString:Connection:private] => mysql:127.0.0.1
    [username:Connection:private] => user1
    [password:Connection:private] => password1
    [lastActionTimeStamp:Connection:private] =>
)
Destructor called
close() called
*/
```

# Magic methods: __toString()

The **__toString()** method allows a class to decide how it will react when it is treated like a string. For example, what **echo $obj;** will print.

```php
<?php

class TempFileManager
{
    private string $fileName;

    public function __construct($userDefinedFileName = '')
    {
        if ($userDefinedFileName != '') {
            $this->fileName = $userDefinedFileName;
        } else {
            $this->fileName = md5(rand());
        }
    }

    public function __toString() : string
    {
        return "The file name is [" . $this->fileName . "]\n";
    }
}

$tempFile = new TempFileManager();
echo $tempFile;
// The file name is [6c4d2b2c2689259911855f2c4a18cd64]
```

Here we define the string representation of our class objects.

# Magic methods: __invoke()

The **__invoke()** method is called when a script tries to call an object as a function. Useful for complex frameworks creation.

```php
<?php
class CallableClass
{
    public function __invoke($x)
    {
        var_dump($x);
    }
}
$someObject = new CallableClass;
$someObject(5);
// int(5)

var_dump(is_callable($someObject));
// bool(true)
```

Yes, it works ☺.

# Magic methods: __set_state()

The **__set_state()** method is called for classes exported by **var_export()** (which returns a parsable string representation of a variable).

```php
<?php

class SomeClass
{
    public string $somePropertyOne;
    public string $somePropertyTwo;

    public static function __set_state($someArray)
    {
        $someObject = new SomeClass;
        $someObject->somePropertyOne = $someArray['somePropertyOne'];
        $someObject->somePropertyTwo = $someArray['somePropertyTwo'];
        return $someObject;
    }
}

$testObject = new SomeClass;
$testObject->somePropertyOne = 'One';
$testObject->somePropertyTwo = 'Two';
```

```php
$executableCode = var_export($testObject, true);
echo $executableCode;
/*
  SomeClass::__set_state(array(
  'somePropertyOne' => 'One',
  'somePropertyTwo' => 'Two',))
*/

eval('$newTestObject = ' . $executableCode . ';');
var_dump($newTestObject);

/*
class SomeClass#2 (2) {
  public string $somePropertyOne =>
  string(3) "One"
  public string $somePropertyTwo =>
  string(3) "Two"
}
```

Here we define the structure of the executable code returned by var_export().

# Magic methods: __clone()

Once the cloning is complete, the newly created object's **__clone()** method will be called to allow any necessary changes to be made.

```php
<?php

class TempFileManager
{
    private string $fileName;

    public function __construct($userDefinedFileName = '')
    {
        if ($userDefinedFileName != '') {
            $this->fileName = $userDefinedFileName;
        } else {
            $this->fileName = md5(rand());
        }
    }

    public function __clone()
    {
        $this->fileName = md5(rand());
    }
}
```

```php
$tempFileOne = new TempFileManager();
$tempFileTwo = clone $tempFileOne;

var_dump($tempFileOne);
/*
class TempFileManager#1 (1) {
  private string $fileName =>
  string(32) "c45ad39511884303df2924b5f01fdcb3"
}
*/

var_dump($tempFileTwo);
/*
class TempFileManager#2 (1) {
  private string $fileName =>
  string(32) "cdd1683525d18df311d9163c7c227f63"
}
*/
```

Otherwise two objects will try to work this the same file.

# Magic methods: __debugInfo()

This method is called by **var_dump()** when dumping an object to get the properties that should be shown.

```php
<?php

class Liar
{
    private $someValue;

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

    public function __debugInfo()
    {
        return [
            'username' => 'user1',
            'password' => 'password1',
        ];
    }
}
```

```php
$liar = new Liar(10);
var_dump($liar);
/*
class Liar#1 (2) {
  public $username =>
  string(5) "user1"
  public $password =>
  string(9) "password1"
}
*/
```

Of course, in real-life situations this approach is useful to prevent unintentional disclosure of sensitive data.

# Interfaces

Interfaces allow to create code which specifies which methods a class must implement, without having to define how these methods are implemented.

Interfaces share a namespace with classes and traits, so they may not use the same name.

```php
<?php

// Interface definition
interface iStorableInDB
{
    public function store();
    public function retrieve();
}

// Interface implementation:
// this class MUST implement
// store() and retrieve() methods.
class LongConnection implements iStorableInDB
{
    // store() and retrieve() methods implementation here
}
```

Remember, that **instanceof** reacts not only to class hierarchy, but to interface implementation also.

A class can implement two (and more) interfaces which define a method with the same name, only if the method declaration in all interfaces is identical.

# Traits

**While a class may extend only one parent object, it may use unlimited traits.**

Traits are a mechanism for code reuse in the situation on single inheritance approach (as PHP does not support multi-inheritance).

Implement once…

```php
<?php

trait tNicePropertiesOutput
{
    public function getNicePropertiesOutput() : string
    {
        $objectVariables = get_object_vars($this);
        $result = '';
        foreach ($objectVariables as $variableName => $variableValue) {
            $result .= '[' . $variableName . '] = [' . $variableValue . "]\n";
        }
        return $result;
    }

}
```

```php
$someObjectOne = new SomeClassOne();
echo $someObjectOne;
/*
[somePropertyOne] = [1]
[somePropertyTwo] = [2]
[somePropertyThree] = [3]
*/

$someObjectTwo = new SomeClassTwo();
echo $someObjectTwo;
/*
[somePropertyOne] = [One]
[somePropertyTwo] = [Two]
*/
```

Use many times!

```php
class SomeClassOne
{
    use tNicePropertiesOutput;

    public int $somePropertyOne = 1;
    public int $somePropertyTwo = 2;
    public int $somePropertyThree = 3;

    public function __toString(): string
    {
        return $this->getNicePropertiesOutput();
    }
}

class SomeClassTwo
{
    use tNicePropertiesOutput;

    public string $somePropertyOne = 'One';
    public string $somePropertyTwo = 'Two';

    public function __toString(): string
    {
        return $this->getNicePropertiesOutput();
    }
}
```

# Attributes

Attributes are meta-data elements added for PHP classes, functions, closures, class properties, class methods, constants, and even on anonymous classes. This data may be used by PHP engine, frameworks and so on…

```php
<?php

// Declaring and using an attribute
#[Attribute]
class LosslessSrializable{}

#[LosslessSrializable]
class SafeToSerialize{}

class UnsafeToSerialize{}

$objects[] = new SafeToSerialize;
$objects[] = new UnsafeToSerialize;

foreach ($objects as $object) {
    $reflection = new ReflectionClass($object);
    $attributes = $reflection->getAttributes();
    foreach ($attributes as $attribute) {
        if ($attribute->getName() == 'LosslessSrializable') {
            echo 'This is an object of [' . $reflection->getName() . '] class. It is safe to serialize.';
        }
    }
}

// This is an object of [SafeToSerialize] class. It is safe to serialize.
```

Of course, real-life implementation is more complex, but the general idea stays the same – you may enrich your code with such capabilities.

# Magic Methods, Interfaces, Traits, Attributes

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