

# Server-side Web Development

## Unit 05. Classes in PHP. Exceptions.

Fidel Oltra, Ricardo Sánchez



IES Jaume II El Just  
Tavernes de la Valldigna  
Departament d'Informàtica  
Curs 2024-25

## Index

<b>1</b>	<b>Object-Oriented Programming</b>	<b>3</b>
<b>2</b>	<b>Classes in PHP</b>	<b>3</b>
2.1	Access modifiers . . . . .	4
2.2	Creating objects . . . . .	4
2.3	Comparing objects . . . . .	5
2.4	Constructor method . . . . .	7
2.5	Destructor method . . . . .	8
2.6	Extends and inheritance . . . . .	8
2.6.1	Constructor method and inheritance . . . . .	9
2.6.2	Method overriding . . . . .	10
2.6.3	Final keyword . . . . .	11
2.7	Const keyword . . . . .	12
2.8	Static modifier . . . . .	13
2.9	Implements and interfaces . . . . .	14
2.10	Traits . . . . .	15
2.11	Abstract classes and methods . . . . .	16
2.12	Traits vs Interfaces vs Abstract Classes . . . . .	17
2.13	Polymorphism in PHP . . . . .	17
2.14	Anonymous classes . . . . .	19
2.15	Magic methods . . . . .	19
2.16	Serializing objects . . . . .	20
2.17	Some useful class functions . . . . .	21
2.18	Some useful object functions . . . . .	21
<b>3</b>	<b>Namespaces</b>	<b>22</b>
<b>4</b>	<b>Handling errors and exceptions in PHP</b>	<b>23</b>
4.1	Exceptions . . . . .	23
4.1.1	How to handle exceptions . . . . .	23
4.1.2	Finally section . . . . .	25
4.1.3	Some exceptions handling methods . . . . .	26
4.1.4	Extending the Exception class . . . . .	28
4.1.5	Specific exceptions . . . . .	28
4.1.6	Throwing an exception . . . . .	29
4.2	Warnings . . . . .	30
4.2.1	How to handle a warning? . . . . .	30

4.2.2	Using a error log file . . . . .	31
-------	----------------------------------	----

## 1 Object-Oriented Programming

In this unit we assume that you are already familiar with the concepts of **Object-Oriented Programming (OOP)**: classes, objects, methods, inheritance and so on.

So, instead of explaining these theoretical concepts, we will focus on practice: how to work with classes and objects in PHP? Let's see it.

## 2 Classes in PHP

A class is defined by using the keyword **class** followed by its name.

```
class myClass {  
    ...properties  
    ...methods  
}
```

As you can see, the properties and methods of the class are declared inside curly brackets ({}).

```
class Person {  
    private $name;  
    private $address;  
}
```

The methods are declared inside the brackets too. Methods are declared using the keyword **function** followed by its name.

Inside the methods, the properties of the class are referenced using **\$this->propertyName**.

```
class Person {  
    private $name;  
    private $address;  
  
    // get methods  
    function getName() { return $this->name; }  
    function getAddress() { return $this->address; }  
  
    // set methods  
    function setName($name) { $this->name = $name; }
```

```
function setAddress($address) { $this->address = $address; }  
}
```

## 2.1 Access modifiers

Properties and methods can be **private**, **protected** or **public**.

We can't access directly to a private method or property outside the class. So we must implement **get** and **set** methods like seen in the example above.

Properties and methods declared as **protected** can only be invoked directly inside the class and inside its child classes.

Properties and method declared **public** can be used directly outside the class and its child classes.

**Properties** and **methods** declared without any explicit visibility keyword are defined as **public**.

```
class Person {  
    private $name;  
    public $address;  
    protected $birthDate;  
    $age; //this is public  
    ...  
}  
  
$onePerson = new Person();  
$onePerson->name = "John Smith"; // FAILS  
$onePerson->address = "Trafalgar Square"; // IT WORKS  
$onePerson->birthDate = "2000-03-03"; // FAILS  
$onePerson->age = 23; // IT WORKS
```

## 2.2 Creating objects

An object is created using the keyword **new** followed by the name of the class and a pair of parenthesis (with or without parameters).

```
$someone = new Person();
```

We have instantiated the class in order to create the object **\$someone**.

Properties and methods are referenced using **->**.

```
class Person {
    private $name;

    function getName() { return $this->name; }
    function setName($name) { $this->name = $name; }
}

...
$someone=new Person(); // we create the object
$someone->name="John Smith";
// we assign a value to the property $name
// this will only work if the property $name is public

$someone->setName("John Smith");
// this assigns a value to the $name property using the public method
// this is how we do it when the property is defined as private

$name = $someone->name;
// this only works if the property name is public

$name = $someone->getName();
// this is how we get the value of a private property
```

We can check if an object belongs to a certain class using the operator **instanceof**.

```
if ($onePerson instanceof Person) {
    echo " The object belongs to the Person class<br/>";
}
else {
    echo " The object doesn't belong to the Person class<br/>";
}
```

## 2.3 Comparing objects

The equal operator (==) returns **True** if the two compared objects are instances of the same class and their properties have the same values and types.

The identity operator (===) returns **True** if the two compared objects reference the same instance of the class.

Let's see an example with this class:

```
class Rectangle {  
    private $base;  
    private $height;  
    function __construct($base,$height) {  
        $this->base=$base;  
        $this->height=$height;  
    }  
}
```

We declare two objects with the same values for base and height:

```
$rectObject01=new Rectangle(5,10);  
$rectObject02=new Rectangle(5,10);
```

Now, let's compare the objects:

```
if($rectObject01==$rectObject02) {  
    echo "The two objects are equal<br/>";  
}  
else {  
    echo "The two objects are not equal<br/>";  
}  
  
if($rectObject01=== $rectObject02) {  
    echo "The two objects are identical<br/>";  
}  
else {  
    echo "The two objects are not identical<br/>";  
}
```

We will get:

Now, let's try again defining one object as a copy of the other.

```
$rectObject01=new Rectangle(5,10);  
$rectObject02=$rectObject01;
```

Now, if we compare the two objects with == and === we will get:

The two objects are equal  
The two objects are not identical

**Figure 1:** Comparing objects

The two objects are equal  
The two objects are identical

**Figure 2:** Comparing identical objects

In this case, `rectObject02` is not an exact copy of `rectObject01`, it is a reference to the same object. If we change any property of `rectObject02`, we are changing the same property of `rectObject01`.

## 2.4 Constructor method

A **constructor method** is a method that, if exists, is invoked every time we create an object. In some languages, the constructor method must have the same name as the class. In PHP we can declare a constructor method defining a method with the name **`__construct`** with or without parameters.

Unlike other languages, in PHP we're not allowed to create two or more **`__construct`** methods with different parameters.

```
class Person {  
    private $name;  
  
    public function __construct($name) {  
        $this->name=$name;  
    }  
    ...  
}  
  
$onePerson=new Person("John");
```



```
// the constructor is automatically invoked when the object  
// is created assigning "John" to $name
```

## 2.5 Destructor method

A **destructor method** is automatically called when the last instance of an object is destructed, or when we stop/abandon the script where it has been created.

In PHP we can create a method **\_\_destruct()** in order to force the destructor to do any task other than just “garbage collecting”.

```
class Person {  
    function __destruct() {  
        // the code goes here  
        // this code will be executed when the object is destroyed  
    }  
}
```

We can manually remove all the references to an object using the **unset** function (like we did with session variables).

```
unset($onePerson);
```

Once the **unset** function destroys all the instances of the object, the destructor method of the class will be invoked.

## 2.6 Extends and inheritance

A class can be an extension of another class. In this case, the extended class will inherit the properties and methods of the main class.

We define an extended class using the **extends** clause.

```
class mySubclass extends myClass {  
    ...own properties  
    ...own methods  
}
```

The **mySubClass** is a child class of **myClass**. Thus, **mySubClass** inherits the methods and properties of **myClass**. Let's see an example:

```
// Parent class
class Figure {
    protected $x, $y;
    function __construct($a, $b) {
        $this->x = $a;
        $this->y = $b;
    }
}

// Child class
class Square extends Figure {
    ... // we don't define the $x and $y properties
    ... // but they're inherited from the parent class
}

$figObject = new Figure(10,5);
$squObject = new Square(10,5);
// as $x and $y have been declared as protected in the
// parent class, they can be directly accessed from the
// child class using ->, but they're not accessible outside
// the parent and child classes
```

A class in PHP can only inherit from one parent class, although we have methods to implement something similar to multiple inheritance.

The parent class must be defined before the child class in the script.

### 2.6.1 Constructor method and inheritance

A child class inherits the constructor method from its parent class. The constructor is automatically invoked when an object (both of parent or child classes) is created.

Therefore, **the child class can have its own constructor method**. In that case, **when an object of the child class is created, only the child class constructor will be invoked** automatically.

If we want to invoke the parent constructor in addition to the child constructor, we can do it by explicitly calling the parent constructor with **parent::\_\_construct**.

```
class Teacher extends Person {  
    private $specialty;  
  
    function __construct($name, $surname, $specialty) {  
        parent::__construct($name, $surname);  
        $this->specialty = $specialty;  
    }  
}
```

When we create a **Teacher** object with:

```
$teacher = new Teacher("Fidel", "Oltra", "Informàtica");
```

The `__construct` of the **Teacher** class will be executed. Then, with:

```
parent::__construct($name, $surname);
```

the parent constructor will be invoked and the name and surname are saved into the object. After that, with:

```
$this->specialty = $specialty;
```

The third parameter of the **Teacher** constructor, the specialty (“Informàtica”), will be saved into the object as well.

### 2.6.2 Method overriding

**Method overriding** consists of declaring a method in the child class with the same name, parameters and return type as a method in the parent class. The code in the **overriding method** (the child method) replaces the code in the **overridden method** (the parent method with the same name, parameters and return type).

Thus, we have two methods that will be invoked exactly in the same way. If the method is invoked by a parent class object, the parent method will be called. If the method is invoked by a child class object, the child method will be called.

If we want to execute both the parent and the child methods, we can call the parent method from the child method using again **parent::method\_name**.

Let's see an example. First, without calling the parent method:

```
class Person {  
    protected function greetings() {  
        echo "Hello, I'm a person<br/>";  
    }  
}  
  
class Teacher extends Person {  
    public function greetings() {  
        echo "Hello, I'm a teacher<br/>";  
    }  
}  
  
$p = new Teacher();  
$p->greetings();
```

Result:

Hello, I'm a teacher

**Figure 3:** Child method

Now, we call the parent class method from the child class method.

```
class Teacher extends Person {  
    public function greetings() {  
        parent::greetings();  
        echo "Hello, I'm a teacher<br/>";  
    }  
}
```

We will get:

### 2.6.3 Final keyword

By using the keyword **final** we prevent a class from being extended, or a method from being overridden.

A final method:

Hello, I'm a person  
Hello, I'm a teacher

**Figure 4:** Child method calling the parent method

```
class MyNormalClass {  
    final function dontOverrideMe() {  
        // the class can be extended, but this method  
        // can't be overridden. If we try, we'll get an error  
    }  
}
```

A final class:

```
final class MyFinalClass {  
    ...  
}  
  
class MyChildClass extends MyFinalClass {  
    ...  
}  
// we'll get an error, because the MyFinalClass class can't be extended
```

**Note:** Properties cannot be declared final: only classes, methods, and constants (as of PHP 8.1.0) may be declared as final. As of PHP 8.0.0, private methods may not be declared final except for the constructor.

## 2.7 Const keyword

By using the keyword **const** we can declare constants within a class.

```
class Product {  
    const IVA = 21;  
}
```

Class constants are **case-sensitive**. However, is highly recommended to use uppercase letters to name constants.

Class constants are accessed from outside the class by using the **class name** followed by the operator **::** and the constant name.

```
echo Product::IVA;
```

Constants declared without any explicit visibility keyword are defined as public.

We don't need to create an object to access a public class constant property.

As of PHP 8.3.0, constants can be typed:

```
interface I {
    const string PHP = 'PHP 8.3';
}

class Foo implements I {
    const string PHP = [];
}

// Fatal error: Cannot use array as value for class constant
// Foo::PHP of type string
```

## 2.8 Static modifier

**Static** properties and methods can be called without creating an object. They are defined by using the modifier **static**.

Static properties are useful when every instance (every object) of the class share the same value for a property. Unlike constants, the values of static properties can be modified: the new value will spread to all instances.

Outside the class static properties and methods are accessed by using the name of the class followed by the operator **::** and then the name of the property or method. If the static property is private, it must be modified through a **static public method**. If a method uses the value but don't change it, it doesn't need to be static. Anyway, if we want to use the syntax `Class::method`, the method should be declared as static.

Inside a static method, static properties are referenced using **`$self::property_name`**.

Let's see an example:

```
class Product {
    private static $iva = 21;
    static public function getIva() { return self::$iva;}
    static public function setIva($iva) { self::$iva = $iva; }
}

$prodObject1 = new Product();
$prodObject2 = new Product();

echo Product::$iva; //this won't work because $iva is private
echo Product::getIva()."<br/>"; // this works
echo $prodObject1->getIva()."<br/>"; // 21
$prodObject1->setIva(22); // we change the iva value from one object
echo $prodObject2->getIva(); // 22, the iva has changed in both objects
```

In child classes, if we need to access a parent static property, we will use the keyword **parent** instead of **self**.

```
class MathThings {
    public static $valueOfPi = 3.14159;
    static public function getPI() {
        return self::$valueOfPi;
    }
}

class ChildMathThings extends MathThings {
    static public function getPI() {
        return parent::$valueOfPi;
    }
}
```

## 2.9 Implements and interfaces

By using the **implements** clause, we declare that the class must implement a certain **interface**. An interface specifies which methods a class must implement, but without developing that methods.

Interfaces are defined in the same way as classes, but with the **interface** keyword instead of **class**, and without defining the contents of any method.

When a class implements an interface, it's mandatory to develop all its methods because they are not developed in the interface. **All the methods in the interface must be public.**

**Abstract classes** can implement only some methods of the interface, but then classes that extend the abstract class must implement the rest of the methods. We will learn about abstract classes a little later.

A class can implement two or more interfaces. To do so, we will separate each interface name with a comma in the class declaration.

Interfaces can be extended by other interfaces using the `extends` operator, as classes do.

```
interface WorkTable {
    public function jumpLine(); // function without code
}
class MyTable implements WorkTable {
    public function jumpLine() {
        // here goes the code
    }
}
```

## 2.10 Traits

As we said before, PHP (like Java and other languages) only supports **single inheritance**. Thus, a class can't extend two (or more) parent classes.

By using **traits**, we can implement something very similar to multiple inheritance: one class (or more classes) can use methods from multiple traits.

A Trait is similar to a class, but only has one or more functions. It is not possible to instantiate or use a Trait on its own.

To use a trait inside a class, we must declare the trait with the keyword **use**.

Let's see an example:

```
trait EnglishMessages {
    function WelcomeEnglish() {
        echo "Welcome!";
    }
}
```



```
trait SpanishMessages {  
    function WelcomeSpanish() {  
        echo "Bienvenido!";  
    }  
}  
  
class Messages {  
    use EnglishMessages;  
    use SpanishMessages;  
}  
  
$message=new Messages();  
$message->WelcomeEnglish();  
$message->WelcomeSpanish();
```

The object `$message` can use functions both from **EnglishMessages** and **SpanishMessages**. So the output will be:

Welcome!  
Bienvenido!

**Figure 5:** Using traits

## 2.11 Abstract classes and methods

PHP has **abstract** classes and methods. An abstract class is kind of mix between interfaces and normal classes. In an abstract class we can define abstract and non-abstract methods, while in the interface all the methods are actually abstract.

**Classes defined as abstract cannot be instantiated** (we can't create an object of an abstract class). Any class that contains at least one abstract method must defined as abstract.

When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child class (just like we do with the interfaces, but only with the methods declared as abstract).

```
abstract class WorkTable {
    abstract function jumpALine();

    public function nonAbstract() {
        // here goes the code
    }
}

class MyTable extends WorkTable {
    function jumpALine(){
        // the code goes here
    }
    // we don't need to redefine the nonAbstract method
    // because is already implemented in the parent class
}
```

## 2.12 Traits vs Interfaces vs Abstract Classes

TRAIT	INTERFACE	ABSTRACT CLASS
Contains only developed methods that can be called in the class that uses the trait	Contains only empty methods that must be developed in the class that implements the interface	Can contain both developed and abstract (non-developed) methods

## 2.13 Polymorphism in PHP

The OOP offers us a useful feature named **polymorphism**. By using it, we can make that objects of different classes give a different answer to the same message (method invoked, basically). We can implement polymorphism with **abstract classes** or **interfaces**.

Polymorphism is very useful when we have a collection of objects of different classes that implements / extends the same interface / abstract class.

Let's see an example.

```
abstract class Person {
    private $name;
```

```
    abstract public function welcome();

    public function setName($name) {
        $this->name=$name;
    }

    public function getName() {
        return $this->name;
    }
}

class EnglishPerson extends Person {
    // we must implement the abstract method in Person class
    public function welcome() {
        return "Hello ".$this->getName()."!";
    }
}

class SpanishPerson extends Person {
    // we must implement the abstract method in Person class
    public function welcome() {
        return "Hola ".$this->getName()."!";
    }
}

$persons=array();
$person1=new EnglishPerson();
$person1->setName("James");
$person2=new SpanishPerson();
$person2->setName("Antonio");
array_push($persons,$person1);
array_push($persons,$person2);
foreach($persons as $person) {
    echo $person->welcome()."<br/>";
}
```

We will get this output:

# Hello James!

# Hola Antonio!

**Figure 6:** Polymorphism in PHP

## 2.14 Anonymous classes

As we have anonymous functions in PHP, since PHP 7 we have **anonymous classes** too. As with anonymous functions, an anonymous class is a class with no name that we can use when only a single object is needed.

```
$obj = new class('Hi') {  
    public $x;  
    public function __construct($a) {  
        $this->x = $a;  
    }  
};  
  
echo $obj->x; // "Hi";
```

## 2.15 Magic methods

Magic methods in PHP are special functions prefixed with `__` (*double underscore*) that allow you to handle events and operations in classes.

The magic methods are: `__construct()`, `__destruct()`, `__call()`, `__callStatic()`, `__get()`, `__set()`, `__isset()`, `__unset()`, `__sleep()`, `__wakeup()`, `__serialize()`, `__unserialize()`, `__toString()`, `__invoke()`, `__set_state()`, `__clone()`, and `__debugInfo()`.

We have already seen the magic methods `__construct()` and `__destruct()`. Let's see another magic method: `__toString()`. This method is used to define how an object should be represented as a string. When an object is treated as a string, such as when using `echo`, the `__toString()` method is automatically called and returns the desired string representation of the object.

```
<?php
class Person {
    public $name;
    public $surname;

    public function __toString() {
        return $this->name . ' ' . $this->surname;
    }
}

$person = new Person();
$person->name = "Jane";
$person->surname = "Smith";
echo $person; // outputs "Jane Smith"
```

[Mora about magic methods](#)

## 2.16 Serializing objects

The PHP function `serialize()` generates a storable representation of a value. When working with objects, this can be useful to store the object into a file, a session variable, and so on. The function `serialize()` returns a string.

To get the original value of the serialized object, we can use the `unserialize()` function. The class definition must be included in order to deserialize the object properly.

[PHP Serialization](#)

### 2.17 Some useful class functions

Function	Description
<code>class_exists(\$className)</code>	Returns TRUE if the class exists, returns FALSE otherwise
<code>get_class_methods(\$className)</code>	Returns an array with the names of the class methods
<code>get_class_vars(\$className)</code>	Returns an array with the names of the class properties and their default values (only if they are accessible from the current scope)

### 2.18 Some useful object functions

Function	Description
<code>is_object(\$var)</code>	Returns TRUE if \$var is an object
<code>get_class(\$obj)</code>	Returns the name of the classe which the object belongs
<code>method_exists(\$obj,\$meth)</code>	Returns TRUE if the object \$obj has the method specified in \$meth
<code>get_object_vars(\$obj)</code>	Returns an array with the object properties and their values (only if they are accessible from the current scope)
<code>get_parent_class(\$obj)</code>	Returns the name of the parent class of the object \$obj (FALSE if there is none)

### 3 Namespaces

**Namespaces** are useful to organize and group classes. Namespaces also allow us to define more than a class with the same name, as long as they are in different namespaces.

We define a namespace using the keyword `namespace`. When used, it must be the first sentence in the PHP file.

Let's see an example:

#### File namespace01.php

```
namespace group01;
class SameName {
    ...
}
```

#### File namespace02.php

```
namespace group02;
class SameName {
    ...
}
```

Although we have defined the different namespaces in different files, we can declare multiple namespaces in the same file.

Now, in another PHP file we can do that:

```
include 'namespace01.php'; // including the external file
include 'namespace02.php'; // including the external file

use group01 as g1; // an alias for the namespace
use group02 as g2; // an alias for the namespace

$object01 = new g1\SameName(); // using external namespace
$object02 = new g2\SameName(); // using external namespace
```

[More about namespaces in PHP](#)

## 4 Handling errors and exceptions in PHP

**Error handling** is an essential part of developing an application. PHP provides a number of tools that we can use to handle any kind of errors.

In this unit we will see how manage them at two levels:

- catching and handling throwable errors
- configuring php to handle other types of errors

Handling exceptions and errors is slightly different in PHP7 compared to previous versions. We will go through those differences in this lesson.

### 4.1 Exceptions

A PHP exception happens when the application tries to perform a task and it's unable to do it.

An exception stops the execution unless we catch it and handle it.

By catching exceptions we can:

- avoid showing undesired error messages to the final user
- prevent the application to be suddenly halted

#### 4.1.1 How to handle exceptions

In PHP5 the exceptions would be handled by using the **exception** class and the structure **try...catch**.

```
try {  
    ...  
}  
catch(Exception $e) {  
    echo $e->getMessage();  
}
```

In PHP5 some internal errors can not be handled by using the **exception** class. In PHP7 we have the **class throwable** instead. It covers both exceptions and internal errors.

Just in case we are not sure if the server supports PHP5 or PHP7, we can include both clauses in our **try...catch** block.



```
try {  
    // Code that may cause an Exception or Error.  
}  
catch (Throwable $t){  
    // Executed only in PHP 7, will not match in PHP 5  
}  
catch (Exception $e){  
    // Only in PHP 5, won't be reached in PHP 7  
}
```

Let's see an example. If we run this code:

```
$number = 10;  
$reverseNumber = 1 / $number;  
echo "<h2>The reverse of $number is $reverseNumber</h2>";
```

**The output:** The reverse of 10 is 0.1

But if we change the value of \$number to 0:

```
$number = 0;  
$reverseNumber = 1 / $number;  
echo "<h2>The reverse of $number is $reverseNumber</h2>";
```

We will get an exception:

**Fatal error: Uncaught DivisionByZeroError: Division by zero**

We can handle the exception by using the **try...catch** block:

```
$number=0;  
try {  
    $reverseNumber=1/$number;  
    echo "<h2>The reverse of $number is $reverseNumber</h2>";  
}  
catch(Throwable $t) {  
    echo "An error happened";  
}
```

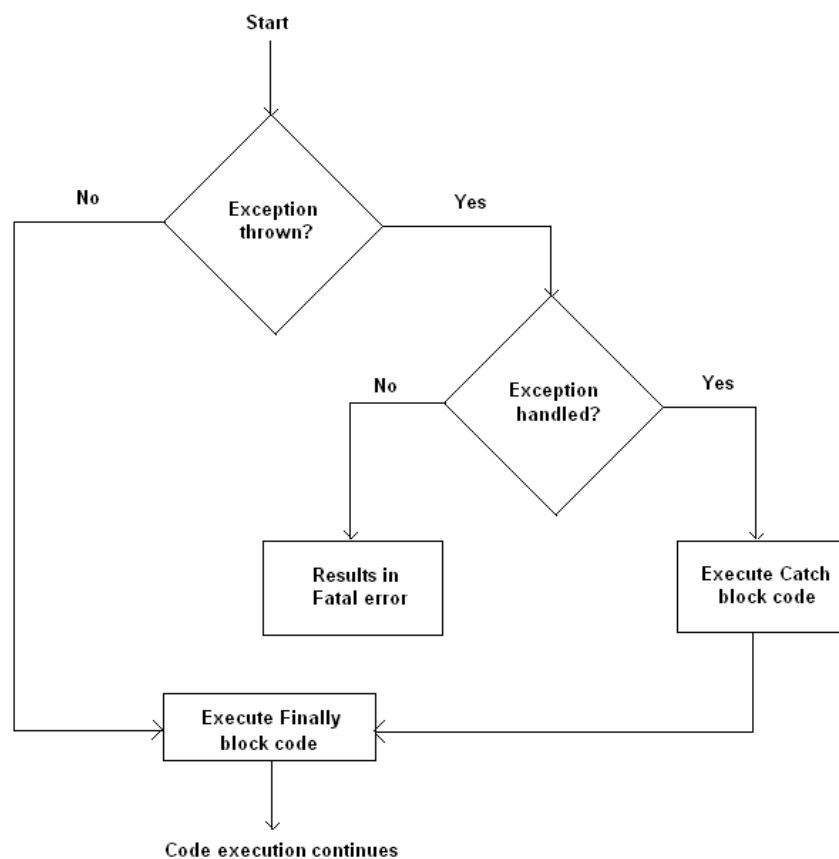
We will get the output:

## An error happened

If an error occurs within the `try` section, the execution will not be halted. It will be redirected to the `catch` section instead. In this section, we might display a message or redirect the application to a safe exit.

### 4.1.2 Finally section

A **finally** section may also be specified after or instead of **catch** clauses. The code inside the finally section will always be executed after the `try` and `catch` blocks, regardless of whether an exception has been thrown, and before normal execution resumes.



**Figure 7:** Exception handling workflow

If a `return` statement is encountered inside either the `try` or the `catch` blocks, the `finally` block will still be executed. Moreover, the `return` statement is evaluated when encountered, but the result will be returned after the `finally` block is executed. Additionally, if the `finally` block also contains a `return` statement, the value from the `finally` block is returned.

```
function reverse($number) {  
    $reverseNumber = 1 / $number;  
    return $reverseumber;    // ERROR: wrong variable name  
}  
  
try {  
    $number=10;  
    echo "<h2>The reverse of $number is".reverse($number)."</h2>";  
}  
catch (Throwable $t) {  
    echo "An error {$t->getMessage()} happened<br/>";  
}  
finally {  
    echo "<h3>This section will run anyway</h3>";  
}
```

**Warning:** Undefined variable \$anverseumber in C:\xampp\htdocs\curs2223\ud08test.php on line 4

## The inverse of 10 is

## This section will run anyway

**Figure 8:** Finally section

### 4.1.3 Some exceptions handling methods

You might want to know more about exceptions. Both **Throwable** and **Exception** provide you with some helpful methods:

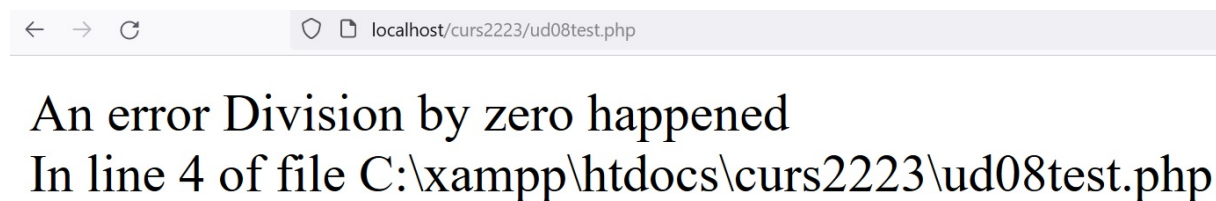
Function	Description
getMessage()	returns the exception message
getCode()	returns the code of the exception
getFile()	returns the name of the file where the exception happened

Function	Description
<code>getLine()</code>	returns the line in the file where the exception happened
<code>__toString()</code>	returns the exception in a String format

An example:

```
$number=0;
try {
    $reverseNumber=1/$number;
    echo "<h2>The reverse of $number is $reverseNumber</h2>";
}
catch(Throwable $t) {
    echo "An error {"$t->getMessage()} happened<br/>";
    echo "In line {"$t->getLine()} of file {"$t->getFile()}<br/>";
}
```

The output:



An error Division by zero happened  
In line 4 of file C:\xampp\htdocs\curs2223\ud08test.php

**Figure 9:** Catching an exception

#### 4.1.4 Extending the Exception class

These methods belong to the PHP predefined **Exception** class. We can define our own custom class to handle exceptions by extending the **Exception** class.

```
class MyException extends Exception {
    // we can override the Exception construct method
    public function __construct($message, $code = 0, Throwable $previous =
        null) {
        // some code
        // ...
        // but then we need to call the parent construct method
        parent::__construct($message, $code, $previous);
    }
    // custom string representation of object
    public function __toString() {
        return __CLASS__ . ": [{".$this->code}]: {".$this->message}";
    }
    // a new function that doesn't exist in the Exception class
    public function customFunction() {
        return "A custom function for this type of exception";
    }
}
```

Now we can call the `$exception->customFunction()` as we do with `getMessage()` and other predefined functions.

#### 4.1.5 Specific exceptions

We can handle specific exceptions by replacing the general clause **throwable** with the name of the specific exception in the catch section.

```
catch (DivisionByZeroError $t)
```

Thus, we can handle different exceptions within the same `try...catch` block:

```
try {
    ...
}
```

```
catch (DivisionByZeroError $t) {  
    ...  
}  
catch (ArithmeticError $e) {  
    ...  
}
```

- [List of predefined exceptions.](#)

#### 4.1.6 Throwing an exception

Exceptions can also be thrown voluntarily, before they occur, with the **throw** command. It is common to do this in functions in order to catch the exception in the main program instead of the function. Thus, we will avoid repetitions of the try...catch loop in each function.

```
function f1() {  
    if (...) {  
        throw new Exception("Message01"); // we throw an exception  
    }  
    return ...  
}  
  
function f2() {  
    if (...) {  
        throw new Exception("Message02"); // we throw an exception  
    }  
    return ...  
}  
  
// with only one try...catch structure we capture exceptions from both  
→ functions  
try {  
    // here we call f1()  
    // and we call f2()  
} catch (Exception $exception) {  
    echo "An exception happened: ".$exception->getMessage();  
}
```

## 4.2 Warnings

Some errors are not throwable. Take a look at this example:

```
function reverse($number) {  
    $reverseNumber = 1 / $number;  
    return $reverseumber;    // ERROR: wrong variable name  
}  
  
try {  
    $number=10;  
    echo "<h2>The reverse of $number is".reverse($number)."</h2>";  
}  
catch (Throwable $t) {  
    echo "An error {"$t->getMessage()} happened<br/>";  
}
```

When we run the script, we will get a **warning**:

**Warning:** Undefined variable \$anverseumber in C:\xampp\htdocs\curs2223\ud08test.php on line 4

# The inverse of 10 is

**Figure 10:** Warning

A **warning**, as you can see, is a non-fatal error that shows a message, but it **doesn't halt the application**.

A warning can't get handled by the **try...catch** structure.

Warnings can be disabled in the **php.ini** configuration file, or by using the function **error\_reporting(e\_error)**. However, handling them is a better practice in developing mode.

### 4.2.1 How to handle a warning?

We can handle warnings by transforming them into exceptions with our own error handling function. First, we create the function:

```
function handleErrors($eLevel, $eMessage, $eFile, $eLine){  
    throw new Exception("Error ".$eMessage." in line ".  
        $eLine." of ".$eFile);  
}
```

And now we convert our function into the error handling function:

```
set_error_handler("handleErrors");
```

We should restore the automatic error handler at the end of the script with the function `restore_error_handler()`.

```
function handleErrors($eLevel, $eMessage, $eFile, $eLine) {  
    throw new Exception("Error ".$eMessage." in line ".$eLine."  
        of ".$eFile); // both warnings and exceptions will be thrown as  
    → exceptions  
}  
  
function reverse($number) {  
    $reverseNumber = 1 / $number;  
    return $reverseNumber; // this will cause a warning  
}  
  
set_error_handler("handleErrors");  
try {  
    $number=10;  
    echo "<h2>The reverse of $number is ".reverse($number)."</h2>";  
}  
catch (Throwable $t) {  
    echo "An error {$t->getMessage()} happened<br/>";  
}  
restore_error_handler();
```

#### 4.2.2 Using a error log file

We can send our error messages to a **.LOG File**. In this case, the name of the file should not be **error.log** because it already exists and is handling apache errors.



```
function handlingErrors($eLevel, $eMessage, $eFile, $eLine) {  
    error_log("$eMessage in $eFile, line $eLine", // message  
        3, // append mode  
        "c:/xampp/apache/logs/user_errors"); // file route/name  
}
```

In addition, we can add the username (`get_current_user()`), the IP (`$_SERVER['remote_addr']`) of the client that launched the script, the date and any other information available.

```
function handleErrors($eLevel, $eMessage, $eFile, $eLine){  
    $newMessage = "Date: ".date("H:i d-m-Y ").$eMessage.  
        " in file ".$eFile." line ".$eLine.  
        " User: ".get_current_user()." from IP: ".  
        $_SERVER['REMOTE_ADDR'];  
    error_log("$newMessage in $eFile, line $eLine",  
        3,  
        "c:/xampp/apache/logs/user_errors");  
}
```