Object Oriented Language (in PHP of course!)

1. Fundamentals

1.1 Encapsulation

Grouping code and data together in logical units (called `classes`). This is also called `Information Hiding`.
--> Divide the applicaiton into separate entities whose internal components can change without altering their external interfaces.

1.2 Class

`Class` = functions (`methods`) + variable (`properties`).
The methods and the properties works together and provide a specific interface to the outside world.

A class is a `blueprint`, this means it cannot be used directly, it must be `instanciated` into `objects`.

1.2.1 Declaring a class

```
class MyClass {
    // here we go...
}

// or following WordPress naming conventions:
class My_Class {
    // here we go...
}
```

1.2.2 Instanciating an object

```
$myObject = new MyClass();
// or
$myObject = new MyCLass;
```

It is important to note that objects are objects are **always** passed by `reference` :

```
// instanciate first object
$myObject = new MyClass();
// instanciate second object?
$mySecondObject = $myObject;
```

In this example, both `$myObject` and `$mySecondObject` will point to the `same object` .

1.2.3 Class Intheritance

This is a `key concept` of OOP. This allows a class to `extend` another class, essentially by adding new methods and properties, as well as overriding existing ones.

```
class A {
    public function test() {
        echo 'A';
    }
    public function func() {
        echo 'func A';
    }
}

class B extends A {
    public function test() {
        echo 'B';
    }
}

class C extends B {
    public function test() {
        parent::test();
    }
}

class D extends C {
    public function test() {
        B::test();
    }
}

$a = new A();
$b = new B();
$c = new C();
$d = new D();

// what will be the output of these calls?
$a->test();
$b->test();
$b->func();
$c->test();
$d->test();
```

1.2.4 Calling a method

To call a method on a object, use the operator `->`:

```
$a = new A();
$a->test();
```

for an object to call a method on itself, use the special variable `$this`.
`$this` is only defined within an object scope, and always points to the object itself.

```
class CallMyOwnMethod {
    public function a() {
        return 'a';
    }

    public function b() {
        // call my own methods
        return $this->a() . 'b';
    }
}
```

2. Class methods and properties

2.1 Constructors and destructors

Constructors and destructors are special class methods that are called on object creation and destruction, respectively.

`Constructors` are useful for initializing an object's properties, or executing a startup procedure such as connecting to a database.
The name of the constructor is always `__construct()`.

A `destructor` is called before the object is destroyed, and is useful for performing cleanup procedure such as disconnecting from a database, delete temporary files, ...
The name of the destructor is always `__destruct()`.

Never assume you will know when an object will be destroyed. An object is destroyed when *all* references to this object are gone.

```
$a = new A();
$b = $a;
unset($a); // destructor is not called because of $b
```

The destructor is called at the end of the script if it hasn't be called before, so it will always be called at some point.
There is no way to determine the order of the destruction of the objects

Example where it could go wrong:

- one object ($a) closes the DB connection
- another object ($b) flushes the data to the DB)

If the first object being destroyed is $a, then destroying $b will result in an error because the database connection has already been closed --> Not always safe to rely on the destructors in PHP.

2.2 Visibility

- public: Resource can be accessed from any scope
- protected: Resource can only be accessed from within the class and it descendants
- private: Resource can only be accessed from with the class
- final: Resource can be accessed from any scope, but cannot be overriden in descendant classes (can only be used on `class` and `method`; for `class`: this means the class **cannot** be extended)

Constructors and destructors are almost always `public` (except in some special cases, e.g.: design patterns `Singleton` and `Factory`)

```
class A {
    public $a = 'aa';
    protected $b = 'bb';
    private $c = 'cc';

    public function __construct() {
        var_dump(get_object_vars($this));
    }
}
class B extends A {
    public function __construct() {
        var_dump(get_object_vars($this));
    }
}
class C {
    public function __construct() {
        $a = new A();
        var_dump(get_object_vars($a));
    }
}

// output?
new A();
new B();
new C();
```

2.3 Constants, Static methods and properties

2.3.1 keyword static

Unlike regular methods and properties, static methods and properties are part of a class itself, and not only available in the scope of one of its instances.

```
class A {
    static $counter = 0;

    public function __construct() {
        self::$counter++;
    }

    public function getCounter() {
        return self::$counter;
    }
}

$a = new A;
echo $a->getCounter(); // output?
$b = new A;
echo $b->getCounter(); // output?
echo $a->getCounter(); // output?
```

2.3.2 Keyword constant

Class constants work the same way as normal constants, except they are scoped within a class. Class constants are public, thus available from all scopes.

```
class A {
    constant CPT_NAME = 'lbbbw_job';
    // ...
}
```

Remark: constants are defined in capital letters, it is not required but it is the norm.

3. Abstract classes and Interfaces

Both abstract classes and interfaces are used to create a series of constraints on the base design of a group of classes.

3.1 Abstract classes

An abstract class essentially defines the basic skeleton of all descending classes.
For example, you can use an abstract class to define the basic concept of `car` :

- It has two doors, and a lock
- It has a method to lock and unload the doors

An abstract class cannot be used directly, it must be extended.

```
abstract class Car {
    private doorState;

    public function lockDoors() {
        $this->doorState = 0;
    }

    public function unlockDoors() {
        $this->doorState = 1;
    }

    // different models have different types of doors :
    // Scissor doors, Gullwing doors, Butterfly doors, ...
    // @see https://en.wikipedia.org/wiki/List_of_cars_with_non-standard_door_desi
gns for many non standar ways of opening a door
    abstract public function openDoor($doorNumber);
    abstract public function closeDoor($doorNumber);
}

class Lamborghini extends Car {
    public function openDoor($doorNumber) {
        // open the door in some fancy way
    }

    public function closeDoor($doorNumber) {
        // close the doors in some fancy way
    }
}
```

Remarks:

- As long as one of the methods of the class is declared `abstract`, you have to use the `abstract` keyword on the class name too.
- A class can only extends **one** class

3.2 Interfaces

An interface is a "common" contract that your classes must implement in order to satisfy certain logical requirements.

For example, you are developing an application that must run on different types of databases (MySQL, Oracle, MSSQL, ...).

```
interface DB_Interface {
    public function insert();
    public function update();
    public function save();
}

class DB_MySQL implements DB_Interface {
    public function insert() {
        echo 'insert into MySQL';
    }
    public function update() {
        echo 'update into MySQL';
    }
    public function save() {
        echo 'save into MySQL';
    }
}

class DB_Oracle implements DB_Interface {
    public function insert() {
        echo 'insert into Oracle';
    }
    public function update() {
        echo 'update into Oracle';
    }
    public function save() {
        echo 'save into Oracle';
    }
}

function load_db_provider($name): DB_Interface {
    switch ($name) {
        case 'mysql':
            return new DB_MySQL;
            break;
```

```
        case 'oracle':
            return new DB_Oracle;
            break;
    }

    return null;
}

$db = load_db_provider('mysql');
$db->insert(); // output?
$db->update(); // output?
$db->save(); // output?
echo ($db instanceOf DB_MySQL) ? 'DB_MySQL' : 'DB_Oracle';

$db = load_db_provider('oracle');
$db->insert(); // output?
$db->update(); // output?
$db->save(); // output?
echo ($db instanceOf DB_MySQL) ? 'DB_MySQL' : 'DB_Oracle';
```

Remarks:

- A class can extend multiple interfaces.
- With the use of the function `load_db_provider()`, we have no idea what will be the class of the object that will be returned, and we actually don't care about it. All we want is an object that fills the contract given by the interface, which is a class that implements the 3 methods `insert()`, `update()` and `save()`.

4. Traits

Trait gives us a nice way to reuse code by basically copying and pasting it behind the scenes.

Let's take the example of a forum project, where we have two classes: `Thread` and `Comment`. Both classes have an owner, and therefore would need a method called `owner()` that would return the owner of the thread/comment.
Because we don't want to duplicate the code in the two classes, and because it doesn't make sense to create an parent class just for this small functionality, we could move this code to a `Trait` and include it in both classes:

```
trait OwnerTrait {
    public function owner() {
        return Owner::where(/*...*/)->first();
    }
}

class Thread {
    use OwnerTrait;

    // ...
}

class Comment {
    use OwnerTrait;

    // ...
}
```

5. Exceptions

5.1 What is an exception

An exception provides an error control mechanism that is more fined-grained than traditional error handling and allows a much greater degree of control.

- An exception is an object that is created (`thrown`) when an error occurs
- An unhandled exception is always fatal (script execution ends)
- Exceptions change the flow of the application
- An exception can be thrown from a object's constructor
- Different types of exceptions can be thrown, and in separate locations of the code

5.2 The Exception class

An exceptiont that is thrown is always an instance of the `Exception` class or from one of its descendants (through inheritance).

The base class Exception has different properties that are almost always filled automatically for you: `$message` , `$code` , `$file` , `$line` and the following methods: `getMessage()` , `getCode()` , `getFile()` , `getLine()` , `getTrace()` , `getTraceAsString()` and `__toString()` .
Only `$message` and `$code` must be provided, the rest is done by the interpreter.

5.3 Throwing an exception

You usually throw an exception when an error occurs:

```
if ($error) {
    throw new Exception("custom error message");
}
```

5.4 Custom exceptions

You can of course create your own exceptions:

```
class MyException extends Exception { }

if ($error) {
    throw new MyException();
}
```

5.5 Handling exception

To handle an exception, you will have to use the a `try/catch` statement:

```
try {
    // ...
    if ($error) {
        throw new Exception("there was an error");
    }
    // ...
} catch (Exception $e) {
    // handle the exception
}
```

The `catch` statement is chainable, which means that you don't have to use nested `try/catch` if you want to handle multiple types of exceptions:

```
class DbException extends Exception { }
class NoDataException extends Exception { }

try {
    // connect to the DB (we don't care about the class of this object, example on
ly)
    $error = $db->connect();

    if ($error) {
        throw new DbException();
    }

    $results = $db->executeQuery();
    if(empty($results)) {
        throw new NoDataException();
    }

    // do something
} catch (DbException $dbe) {
    // do something if DB connection isn't working
} catch (NoDataException $nde) {
    // do something else if there is no data
} catch (Exception $e) {
    // do something in case of unexpected error
}
```

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

```
try {
    $db->connect();

    $user = $db->from('users')->where('id', 1)->get();

    // do something with the user data...

    $db->from->('users')->where('id', 1)->update($data);
} catch (Exception $e) {
    Monolog::log('There has been an error while updating the user ID 1');
} finally {
    // make sure the connection to the DB is closed correctly
    if ($db->connected()) {
        $db->disconnect();
    }
}
```

Here is a nice example showing how `finally` works:

```
<?php
function bar1() {
  print "bar1 called\n";
  return 1;
}

function bar2() {
  print "bar2 called\n";
  return 2;
}
function bar3() {
  print "bar3 called\n";
  return 3;
}

function foobar() {
  try {
    throw new Exception();
    return bar1();
  } catch (Exception $e) {
    print "Exception!\n";
    return bar2();
  } finally {
    print "Finally called!\n";
    return bar3();
  }

  return -1;
}

print foobar() . "\n";
```

What will be the output of the above code?

6. Lazy Loading

Instanciating an undefined class will cause a fatal error. This means that you need to include all of the files that you *might* need, rather than loading them as they are needed. This can before very complicated with big projects.

Luckily, PHP features an `autoload` facility that makes it easy to implement `lazy loading` (loading a class on demand).
When referencing a non-existent class, PHP wil try to call the `__autoload()` global function so that the script may be given an opportunity to load it. If after the call to this method, the class is still not defined, the interpreter throws a fatal error.

```
function __autoload($class) {
    $filename = dirname(__FILE__) . '/include/classes/' . strtolower($class));

    if (file_exists($filename) {
        include ($filename);
    }
}

$myObject = new SomeClass();
```

1. `SomeClass` isn't defined, so PHP interpreter will execute `__autoload('SomeClass.php')`
2. This function will try to load the file `include/classes/someclass.php`

    1. The file is found, is included and everything continues normally
    2. The file does not exists, PHP interpreter gives up and throws a fatal error

The problem comes when you include different libraries and they all use their own autoloaders.
The Standard PHP Library (SPL) offers a simpler solution to this problem by allowing you to stack autoloaders using `spl_autoload_register()`.

```
spl_autoload_register('my_autoload');

function my_autoload($class) {
    $filename = dirname(__FILE__) . '/include/classes/' . strtolower($class));

    if (file_exists($filename) {
        include ($filename);
    }
}
```