OOP key concepts by example

Verfasser: Hong Le Nguyen, last update 10.24

1)
OOP constructs

Class, object, object type --

A class is a blueprint for creating objects.

```
class Person {
   // properties
   public function __construct(string $name, string $email) { … }  // constructor
   // methods
}
```

Object is an instance of a class, created using the new keyword . The constructor method is called automatically when an object is created.

```
$person1 = new Person("peter", "mueller@web.de") ;  // object
```

The instanceof Keyword –
allows us to check if an object is an instance of a class

```
if ($student1 instanceof Student) echo "Yes ";
```

Inheritance --

The child-/sub- class (Student) inherits the code of the parent-/base- class (Person).
A subclass can have its own properties and methods.

```
class Student extends Person {
   private $uni; // own properties
   public function __construct(string $name, string $email, string $uni){
      parent::__construct($name, $email);  // Invoking parent constructors
      $this->uni = $uni;
   }
   // own methods
}
```

abstract class --

cannot be instantiated. It provides an interface for other classes to extend.

An abstract method doesn't have an implementation. If a class contains one or more abstract methods, it must be an abstract class.
A class that extends an abstract class needs to implement the abstract methods of the abstract class.

```
abstract class Dumper {
   abstract public function dump($data);
}
class WebDumper extends Dumper {
   public function dump($data) {
      echo "hello $data"
   }
}
```

Interface --
an interface specifies what a class can do, not how to do it.

```
interface PrintAble {  // interface class
   public function printMySelf();
   public function printMySelfOnConsole();
}
```

A class that implements an interface must implement all interface methods - they are public.

```
// src/Entity/Student.php
<?php
namespace Demo\Entity;
use Demo\Entity\Person;
use Demo\Entity\PrintAble;

class Student extends Person implements PrintAble {
   …
   public function printMySelf(){
      // student-specific  implementing code
   }
   public function printMySelfOnConsole(){
      // student-specific  implementing code
   }
}
```

which types the object $student can have ?

```
// test_app.php
<?php
require_once __DIR__ . '/vendor/autoload.php';
use Demo\Entity\Student;
use Demo\Entity\Person;
use Demo\Entity\PrintAble;

$student = new Student('Anna', 'anna@yahoo.de', 'Uni Muen');
if ($student instanceof Student) echo "Yes ";
if ($student instanceof Person) echo "Yes ";
if ($student instanceof PrintAble) echo "Yes";
```

=> output

```
Yes Yes Yes
```

2)
Using Interfaces to write more generic and reusable code

PHP supports *the polymorphism concept* --
objects of different classes to be treated as instances of a common interface or superclass
It enables methods to be invoked without neccearily knowing the exact type of the object.

Benefits of polymorphism --
Polymorphism enables dynamic binding at runtime based on the actual type of the object being referenced.
By using polymorphism, you can reduce coupling and increase code reusability.
Polymorphism promotes code extensibility and maintainability, as new classes that relies on the common interface or superclass can be added without affecting existing code.

To implement polymorphism in PHP,
you can use either *interfaces* or *abstract classes* or *overriding* parent class's method in the child class.

PHP polymorphism using an interface --

Let's take a look at a practical example
We know that caches need to read and write data, and we know that they use keys to identify the cached content. Now we've created an interface :

```
<?php
interface Cache  {
   public function write(string $key, $value): void;
   public function read(string $key);
}
```

We should type-hint that in the functions or constructors of classes that make use of the cache. Here's an example:

```
<?php
class Articles  {
   private $cache;
   public function __construct(Cache $cache) {
      $this->cache = $cache;
   }
   public function fetch() {
      if ($articles = $this->cache->read('articles')) {
         return $articles;
      }
      // Fetch from original non-cached data store and return.
   }
}
```

By *type-hinting* the Cache in the constructor, we ensure that you can only instantiate the Articles class by providing a class instance that implements the Cache interface.

Right, let's create two implementations of our Cache interface.

```
<?php
class RedisCache implements Cache  {
   public function write(string $key, $value): void {
      // Write cache data to Redis.
   }
   public function read(string $key) {
      // Read cache data from Redis.
   }
}
class MongoCache implements Cache  {
   public function write(string $key, $value): void {
      // Write cache data to MongoDB.
   }

   public function read(string $key) {
      // Read cache data from MongoDB.
   }
}
```

Imagine we have a configuration of which cache should be used for our application

```
<?php
return [
   'cache' => RedisCache::class
];
```

we need a config loader function config($key) that reads entries from the configuration.

Let's write some code that will use the Articles class we wrote earlier to fetch a list of articles.

```
<?php
$cache = new config('cache');      // Create our cache driver.
$articles = new Articles($cache);  // Inject it into a new Articles instance.
$articlesList = $articles->fetch();  // Load our article list.
```

The first line creates a new cache instance using the class that we set in our configuration under the 'cache' key.
In the second line, we instantiate a new Articles class, injecting our cache instance into it.
Finally, on the third line, we use the fetch() method to populate an article list.

You see, the above code allows switching to a different cache without having to change the client code (class Articles) thanks to coupling via interface between modules.

3)
Exception Handling

Unexpected errors are called *exceptions*. Exceptions can be attempting to read a file that doesn't exist or connecting to the database server that is currently down.
Instead of halting the script, you can handle the exceptions gracefully. This is known *exception handling*.

The *throw* statement
The throw statement defines a function or method *to throw an exception*. consider the following example:

```
function writeDateInFile(): void {
   $result = file_put_contents("date", date("Y-m-d"));
   if ($result === false) {
      throw new FileWriteException("There was a problem writing file 'date'");
   }
}
```

When PHP encounters a throw statement, it immediately halts the code execution. Therefore, the code after the throw statement won't execute.

The *try...catch* Statement
To handle the exceptions, you use the try...catch statement. Here's a typical syntax of the try...catch statement:

```
try {
  // code that can throw exceptions
}
catch(Exception $e) {
  // code that runs when an exception is caught
}
```

The try...catch statement separates the program logic and exception handlers.

```
try {
   writeDateInFile();

} catch (\FileWritingException $e) {
```

```
        // Do some specific stuff
        …
        // re-throwing the exception (option)
        //  throw new \Exception('some your infos ... ' . $e->getMessage());
    }
```

4)

Package, Namespace and autoloading

Package --
it's common practice to place the related classes called "Package" in a directory and each class in a separate file.

Project organization
An example of a directory structure, your classes are placed under src directory, file Customer.php belongs to package/directory Model.

```
project
    index.php
    composer.json
    src
        Model   // package directory
                Customer.php
```

Namespaces --
help you avoid any potential *name collisions* and provide a way to group related classes.
Each directory (beginning at e.g. src directory) is assigned a namespace, and all classes in the directory have to be marked with the namespace.

An example, assigning namespace "Store" to src directory, namespace "Store\Model" to src/Model directory. The class Customer belongs to package/directory Model, it should be marked with the namespace "Store\Model" at the top.

```
// src/Model/Customer.php
?php
namespace Store\Model;

class Customer {
    ...
}
```

Note:
It's customary to assign the src directory the "Store" namespace. And you can replace Store with a other name, e.g., "Apple\Store".
It's a good practice to imitate the directory structure with the namespace to find classes more easily.

To use a class that belongs to a namespace
    you need to include the file and
    you need to use *the fully qualified name* that includes the namespace
An example, class Customer has the fully qualified name : "Store\Model*Customer*" in clien code

```
<?php
require 'src/Model/Customer.php';
$customer = new Store\Model\Customer('Bob');
echo $customer->getName();
```

or you can import the namespace with the "use" operator like this:

```
<?php
require 'src/Model/Customer.php';
use Store\Model\Customer;
$customer = new Customer('Bob');
echo $customer->getName();
```

Autoloading --
Php allows you to define a special __autoload function that is automatically called whenever a class is referenced. *You can eliminate the need to manually include each class file.*

Composer autoload with PSR-4
PSR-4 is auto-loading standard that describes the specification for autoloading classes including *how namespaces map to the directory structure*.

The Composer tool can generate the autoloader for you according to the PSR-4 specification. You only need to specify the namespace (e.g. Store) for the "root" directory (e.g. src) where your classes are located.

```
// composer.json
{
   "autoload": {
      "psr-4": {
         "Store\\": "src/"
      }
   },
}
```

run the command to generate the autoload.php file.

```
composer dump-autoload
```

The autoload file is added to application at beginning line for using :

```
// index.php
require_once __DIR__ . '/vendor/autoload.php';
```

An important question: with which namespace should e.g. the class Customer be marked ?

Location of class Customer : *src/Model/Customer.php*
according the PSR-4 specification, the namespace should be: *Store\Model*
=>
General namespace form based on location : **root-namespace\[sub-directory-name\]**

An other example, class *src/Service/Payment/Paypal.php* should be marked with namespace *Store\Service\Payment*

5)
Unit Testing

Unit Testing –
In basic terms, unit testing means that you break your application down to its simplest pieces and test these small pieces to ensure that each part is error free (and secure).

PHPUnit
You can perform unit testing in PHP with PHPUnit, a programmer-oriented testing framework for PHP.

PHPUnit Installation –

```
// composer.json
{
   "require-dev": {
      "phpunit/phpunit": "^11.4"
   },
   "autoload": {
      "psr-4": {
         "Hong\\UnitTesting\\": "src/"
      }
   },
}
```

then run the command:

```
composer install
```

How to Write Tests in PHPUnit ? –

The test class, e.g. UserTest, will usually inherit the PHPUnit\Framework\TestCase class.

```
// tests/UserTest.php
<?php
namespace Hong\UnitTesting;
use PHPUnit\Framework\TestCase;
use Hong\UnitTesting\Model\User;

class UserTest extends TestCase {
}
```

Individual *tests* are public methods named with test as a prefix. For example, to test a addFavoriteMovie method on the User class, the method will be named testAddFavoriteMovie.

```
// src/Model/User.php
<?php
namespace Hong\TestProject;
class User {
   public string $name;
   public int $age;
   public array $favorite_movies = [];

   public function __construct(string $name, int $age) { … }
   public function addFavoriteMovie(string $movie): bool { … }
   public function removeFavoriteMovie(string $movie): bool { … }
   …
}
class UserTest extends TestCase {
     …
      // Test addFavoriteMovie
     public function testAddFavoriteMovie(){ ...}
      // Test removeFavoriteMovie
      public function testRemoveFavoriteMovie(){ ... }
}
```

Inside the test method, say testSayHello, you use PHPUnit's method like assertSame to see that some method returns some expected value.

```
// Test addFavoriteMovie
public function testAddFavoriteMovie(){
   $user = new User('John', 18);
   $user->addFavoriteMovie('Avengers');
   $this->assertContains('Avengers', $user->favorite_movies);
}
```

How to Run Tests in PHPUnit ? –

You can run all the tests in a directory using the PHPUnit binary installed in your vendor folder.

```
$ php ./vendor/bin/phpunit tests
```

You can also run a single test by providing the path to the test file.

```
$ php ./vendor/bin/phpunit tests/UserTest.php
```

Project code on Git : https://github.com/hong1234/Php8UnitTest1024

6)
Database Connection

The DbConnect class uses the Singleton pattern to ensure that there is only one instance of the database connection object throughout the application. The getInstance() method creates and returns the instance of the DbConnect class, and the getConnection() method returns the connection object.

```
<?php
namespace Hong\Dao;
use PDO;
class DbConnect {
   private static $instance;
   private $connection;
   private function __construct() {
      try {
       $db = new PDO(
          'mysql:host=127.0.0.1;port=3306;dbname=hongtest',  // connect string
          'root',          // user name
          'Vuanh--66' // password
       );
       $db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
      } catch (\PDOException $e) {
       throw new \Exception('Connection failed: ' . $e->getMessage());
      }
      $this->connection = $db;
   }
   public static function getInstance() {
      if (!isset(self::$instance)) {
         self::$instance = new self;
      }
      return self::$instance;
   }
   public function getConnection() {
      return $this->connection;
   }
}
```

To use the DbConnect class, we simply call the getInstance() method, which returns the same instance of the class every time it is called:

```
<?php
namespace Hong\Dao;
abstract class BaseDao {
   protected final function getDb(){
      $instance = DbConnect::getInstance();
      $conn = $instance->getConnection();
      return $conn;
   }
}
```

The DB connection is ready for use in DAOs

```
<?php
namespace Hong\Dao;
use Hong\Entity\User;

class UserDao extends BaseDao {
   private $db = null;
   public function __construct() {
      $this->db = $this->getDb();
   }

   public function insert(User $user): bool {
      $values = $this->toArray($user); // array
      $fields = array_keys($values);
      $vals   = array_values($values);
      $arr = array();
      foreach ($fields as $f) { $arr[] = '?'; }

      $sql = "INSERT INTO users ";
      $sql .= '('.implode(',', $fields).') ';
      $sql .= 'VALUES ('.implode(',', $arr).') ';

      $statement = $this->db->prepare($sql);
      foreach ($vals as $i=>$v) { $statement->bindValue($i+1, $v); }

      return $statement->execute();
   }
    ...
}
```

Ein Php Framework für Web Applikation

Verfasser: Hong Le Nguyen, last update 08.24

Das Framework implementiert Routing-M-V-C Muster für Web-Anwendung.
Der hier dargestellte Code ist nur auf Wesentliches reduziert.
Der komplette Code auf Git zu finden https://github.com/hong1234/php7MvcFramework

Routing ---

Wie ein (http-)Request mit dem Url (z.B "book/3/order") in ein Method-Aufruf abgebildet wird ?
Dazu dient *ein Route* als ein Abbildung-Regel und z. B so konfiguriert

```
// config/routes.json
...
"book/:id/order": {
   "controller": "Book",
   "method": "order",
   "params": { "id": "number" }
},
```

Routen werden in array gelesen

```
$json = file_get_contents(__DIR__ . '/../../config/routes.json');
$routes = json_decode($json, true);
```

Der Router sucht einen passenden Route für den URL (route matching)

```
// src/Core/Router.php
function route(Request $request){
   ...
   $url = explode('?', $_SERVER['REQUEST_URI'])[0];      // z. B  $url = "book/3/order"

   foreach ($routes as $route => $info) {
      $regexRoute = $this->getRegexRoute($route, $info);
      if (preg_match("@^/$regexRoute$@", $url)) {
         return $this->callHandler($url, $route, $info, $request);
      }
   }
```

Nachdem Route für den Url gefunden ist, hat der Router Informationen um ein (Controller-)Objekt zu erzeugen und den Method zuständig für die Aktion mit Parametern aufzurufen.

```
// src/Core/Router.php
function callHandler($url, $route, $info, $request){
   ...
   $controllerName = '\Bookshop\Controllers\\'. $info['controller'] . 'Controller';
   $controller = new $controllerName($this->di-container, $request);
   $params = $this->extractParams($url, $route);
   return call_user_func_array([$controller, $info['method']], $params);
```

Das Ergebnis ist an View-Engine übergegeben. Der View-Engine erzeugt Html als Response an Client.

Dependency Injection ---

Wir stellen Dependency-Container- und Request-Objekt in den Eigenschaften des Controller-Objekt zur Verfügung.

```
// src/Controllers/BookController.php
class BookController extends AbstractController {
   ...
   function __construct(DIContainer $di-container, Request $request){
      parent::__construct($di-container, $request);
      ...
   }
```

Der Method in BookController kann Dependencies/ (Database-manager-, View-Engine-, Request-) Objekte wie folgend holen

```
function order($id): string {
   $database-manager = $this->di-container->get('doctrine_manager');
   $view-engine = $this->di-container->get('twig_manager');
   $data = $this→request->getParam('param-name');
```

Code-Organisation und Autoloading ---

Code sind in Pakete unter Verzeichnis src/…/ und in Namespaces Bookshop\...\ organisiert.
Ein Autoloader für den Code wird anhand Definition durch Composer erzeugt

```
// composer.json
{
   "autoload": {
      "psr-4": { "Bookshop\\": "src/" }
   },
}
```

Alle Autoloaders (inklusive Autoloaders für Bibliotheken) befinden sich in Datei /vendor/autoload.php
Config-Dateien, Templates für View-Engine sind unter Verzeichnis /config und /views

Booting und Http-Request Processing ---

```
// index.php

// Autoloaders bekannt machen
require_once __DIR__ . '/vendor/autoload.php';

$di = new DIContainer();              // DI-Container erzeugen
$di->set('alias_name', $instance);   // $di initialisieren
$router = new Router($di);            // Router erzeugen

// Request behandeln und Antwort zu Client senden.
$response = $router->route(new Request());
echo $response;
```