CURSO PHP ORIENTADO A OBJETOS

¿Que es un objeto?

Pensar en objetos es muy parecido a como vemos el mundo, si observamos a nuestro alrededor todo esta compuesto por objetos, sea una casa, un carro, una hoja, etc. Cada objeto está compuesto por dos cosas fundamentalmente, sus propiedades y sus comportamientos. Por ejemplo, podemos decir que un Teléfono tiene propiedades como color, modelo o compañía telefónica, y tiene comportamientos como realizar llamadas, envío de mensajes de texto, entre otros. Podemos utilizar esto como base para la Programación Orientada a Objetos, transformando un elemento de la vida real en elementos utilizables dentro de la programación.

Como ya es sabido, un objeto aunque sea el mismo, puede tener distintos comportamientos o propiedades, continuando con el ejemplo anterior, el objeto puede ser un Teléfono, pero existen una gran cantidad de teléfonos de muchas marcas, colores, tamaños y características que lo hacen distinto a los demás, esto define lo que es la identidad del objeto, porque aunque comparten características similares, como las llamadas, encendido, apagado, entre otros, cada uno tiene algo que lo hace diferente.

Los objetos dentro de la programación son muy parecidos a la realidad. Pues que un objeto en programación tiene

propiedades (o también llamados atributos) y comportamientos (que serían las funciones también llamadas métodos).

Las Clases

Podemos ver las clases como una abstracción de un elemento de la vida real para transformarlos en elementos utilizables dentro de la programación. Las clases están conformadas tanto por propiedades (que son como variables dentro de la clase) como por métodos (que son como funciones dentro de la clase).

Como se menciono anteriormente, las propiedades las podemos comparar con las variables dentro de la clase, en el ejemplo del vídeo tenemos como propiedades de la Clase Persona el nombre y el apellido

```
var $firstName, $lastName;
```

Un método es una acción realizada por un objeto, estos están definidos como las funciones de la clase. Por ejemplo el método **fullName** me retorna el nombre completo de una Persona.

```
function fullName()
{
    return $this->firstName . ' ' . $this->lastName;
}
```

Se puede decir que las clases son solo plantillas cuyos métodos o acciones no serán ejecutados en tanto no exista un objeto que requiera su aplicación.

Para crear una nueva instancia de un objeto tenemos la palabra reservada new:

```
$person1 = new Person('Duilio','Palacios');
```

Ahora veamos que funcion cumple el método constructor:

```
1 function __construct($firstName, $lastName)
2 {
3    $this->firstName = $firstName;
4    $this->lastName  = $lastName;
5 }
```

Dentro del código de un constructor se suelen asignar los valores de algunas o todas las propiedades de dicho objeto, para conseguir que el objeto sea creado con valores iniciales. Por ende, un método constructor se ejecuta cada vez que se instancia la clase. No es de carácter obligatorio contar con uno, pero suelen ser bastante útiles. En nuestro ejemplo, cuando se inicia un objeto del tipo Persona va asignar a las propiedades $firstName y $lastName con los datos que coloquemos cuando instanciamos nuestro objeto.

```
1 $person1 = new Person('Duilio', 'Palacios');
```

Para asignar valores o referirnos a la propiedad dentro de una clase se utiliza **$this**, por ejemplo si utilizamos **$this-**

>**firstName** nos referimos a la variable dentro de nuestra clase Persona.

Cada vez que se instancia una clase se crea un objeto nuevo:

```
1 $person2 = new Person('Ramon','Lapenze');
```

Con esto creamos dos objetos del tipo persona, y cada uno tiene su propio nombre y apellido. De esta manera podemos evitar la repetición de código.

Se puede acceder a los métodos de manera muy fácil, se utiliza la fecha (->) la cual nos permite acceder a las propiedades o métodos de nuestra Clase.

```
1 echo "{$person1->fullName()} es amigo de {$person2->fullName()}";
```

Ejemplo

Hagamos un ejercicio con un Teléfono, que en este caso sería el objeto de la clase, algunas de las características pueden ser modelo, color y compañía, por otro lado dentro de sus funciones tenemos realizar llamadas y enviar mensajes. Partiendo de eso, podemos crear nuestro objeto:

```
1  /**
2   * Clase Telefono
3   */
4  class Phone
5  {
6      var $model;
7      var $color;
8      var $company;
9
10     function __construct($model, $color, $company)
11     {
```

```
12        $this->model   = $model;
13        $this->color   = $color;
14        $this->company = $company;
15    }
16
17    function call()
18    {
19        return 'Estoy llamando a otro móvil';
20    }
21
22    function sms()
23    {
24        return "Estoy enviando un mensaje de texto";
25    }
26 }
27
28 $nokia = new Phone('Nokia', 'Blanco', 'Movistar');
```

Como se pudo observar, las características son parte de las propiedades del objeto, definidas en variables, y las funciones son las acciones o métodos que dicho objeto puede realizar. Tenemos que utilizar el símbolo `->` para interactuar con dichas propiedades y métodos:

```
1 echo $nokia->call(); // Estoy llamando a otro móvil
2 echo $nokia->color; // Imprimirá Blanco
```

Ejercicios Propuestos

1. Realiza una definición de algún objeto de tu preferencia, por ejemplo: Un carro. Tengo un Toyota de color rojo, con cuatro puertas el cual puede acelerar y tocar la bocina.
2. A partir del ejercicio anterior, procede a crear una Clase con las propiedades y los métodos que consideres que estén dentro de tu definición.

Clase 2: Encapsulamiento, getters y setters en PHP

Hoy vamos a hablar sobre uno de los objetivos más importantes de la Programación Orientada a Objetos, el cuál es poder proteger y ocultar información, a dicho proceso se le denomina **encapsulamiento**.

El encapsulamiento de datos previene que el desarrollador haga cambios inesperados al sistema, como también ocultar la información para que no pueda ser modificada o vista por otras clases y esto es muy útil pero además fácil de hacer, como aprenderemos en la lección de hoy.

¿Cómo accedemos a dicha información de forma segura? En el desarrollo de esta clase vamos a explicarte cómo podemos modificar de forma segura las propiedades (o variables) de una clase y cómo podemos presentar dicha información al usuario de una forma correcta con el uso de métodos conocidos como **Setters** y **Getters**.

Recuerda tener siempre en cuenta:

Getter: Su función es permitir el obtener el valor de una propiedad de la clase y así poder utilizar dicho valor en diferentes métodos.

Setter: Su función permite brindar acceso a propiedades especificas para poder asignar un valor fuera de la clase.

El encapsulamiento en la programación orientada a Objetos ¿Recuerdan que este también es un elemento fundamental del modelo de objetos? sabía que esa era la respuesta, entonces empezaremos con la lección.

Definamos Encapsulamiento:

Es el proceso de almacenar en una misma sección los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar el interfaz contractual de una abstracción y su implantación.

Pero..! ¿Cómo consigo esto?, ¡tranquilo! esto se consigue a través de la ocultación de información, ves como todo es más fácil estando relajado, ¿No sabes que es la ocultación de información?, *¡Tranquilo!, No, aún no estás tranquilo, ¿Ya?* seguimos, Ocultación de información es el proceso de ocultar “Todos los Secretos” de un objeto que no aportan a sus características específicas.

Para que la abstracción funcione como debe, la implementación debe estar encapsulada, nunca está de más recordar que cada clase debe tener dos partes, una

interfaz y una implementación, tranquilo no te asustes si no sabes que es una clase, ya llegaremos a ese tema, de momento manténgase concentrado en la encapsulación.

Existen tres niveles de acceso para el encapsulamiento, los cuales son:

Público (Public): Todos pueden acceder a los datos o métodos de una clase que se definen con este nivel, este es el nivel más bajo, esto es lo que tu quieres que la parte externa vea.

Protegido (Protected): Podemos decir que estás no son de acceso público, solamente son accesibles dentro de su clase y por subclases.

Privado (Private): En este nivel se puede declarar miembros accesibles sólo para la propia clase.

Voy a hacer un pequeño ejemplo, no usaré ningún lenguaje de Programación Orientado a Objetos, debido a que el Curso es Programación Orientada a Objetos en "General".

El Ejemplo del Vehículo nuevamente, Usaremos la característica COLOR.

Contexto 1: Se necesita que cualquiera pueda acceder a el color de un vehículo, entonces:

Opción a: *Declaro entonces COLOR como Privado*

Opción b: *Declaro entonces COLOR como Protegido*

Opción c: *Declaro entonces COLOR Como Público*

¡Claro!, Sí escogiste la Opción C, te has Ganado un pase al siguiente ejemplo…, de lo contrario vuelve a darle un repaso a los niveles de acceso.

Contexto 2: Se necesita qué color se pueda acceder a través no sólo de vehículo, sí no ahora también de Buses, y como todos sabemos un bus es un tipo de vehículo, entonces también deberá tener acceso a color.
Opción a: *Declaro entonces COLOR como Privado*
Opción b: *Declaro entonces COLOR como Protegido*
Opción c: *Declaro entonces COLOR Como Público*
Correcta Opción b

Contexto 3: Se necesita que color se pueda acceder solamente para vehículo.
Opción a: *Declaro entonces COLOR como Privado*

Opción b: *Declaro entonces COLOR como Protegido*

Opción c: *Declaro entonces COLOR Como Público*

Correcta Opción a

Sesion 3: Herencia y abstracción con PHP

La herencia en programación orientada a objetos, nos permite tener clases que extiendan de otras, heredando así sus propiedades y métodos no privados. En teoría esto puede sonar complicado, pero en la práctica es muy sencillo como vamos a aprender en la clase de hoy.

Notas
Recuerda que una clase puede heredar de otra usando la palabra reservada ***extends*** para heredar de una clase:

```
Class child extends Parents{
//
}
```

Una clase Padre solo puede heredar sus variables y métodos con visibilidad publica (**public**) o protegida (**protected**) y no privada (**private**) esto lo veremos con más calma más adelante.

Para crear una clase Abstracta debes de utilizar la palabra reservada **abstract** antes de la palabra **class**

```
abstract class Parents {
  //
}
```

Una clase abstracta solo puede heredar sus variables y métodos con visibilidad publica (**public**) o protegida (**protected**) ademas de sus métodos abstractos(**abstract**).

```
abstract class Parents {
   abstract protected function foo()
   {
     //
   }
   abstract public function bar()
   {
     //
   }
}
```

Las clases que heredan de otra clase pueden cambiar el comportamiento de la clase padre sobreescribiendo sus métodos

```
class Parents {


   public function say()
   {
     echo 'Hola!';
   }
}
class Child extends Parents {
   public function say()
   {
     echo 'Hola Mundo!';
   }
}
$child = new Child();
$child->say(); // Hola mundo;
```

Finalmente hay que acotar que PHP no permite la herencia multiple, es decir que no puedes utilizar extends de la siguiente forma:

```
class Chind extends Parent, Other {
   //Esto no es válido!!!!
}
```

Sesion 4: Interacción entre objetos

Una característica muy importante de la programación orientada a objetos es la capacidad que los objetos tienen para interactuar con otros. En programación estructurada nuestro código se lee de arriba hacia abajo y escribimos procedimientos de hasta cientos de líneas. En OOP dividimos las responsabilidades de un procedimiento en pequeñas clases y métodos y logramos que un método interactue con otros. De esta manera aunque ya no será posible leer nuestro código en línea recta, podremos hacer cambios más fácilmente en el sistema, escribir pruebas, reusar código, etc.

Ahora bien aunque hayas aprendido a declarar clases y métodos, es muy posible que sigas teniendo el paradigma de la programación estructura por mucho tiempo y que cometas ciertos errores que afecten la calidad de tu proyecto y no te permitan aprovechar los beneficios de la OOP, es por ello que en esta clase te enseñaré cómo puedes diseñar tus métodos para lograr una interacción correcta entre objetos, siguiendo el principio **“tell, don’t ask”**. Además aprenderemos sobre la declaración de tipos en PHP y veremos un repaso de herencia, getters y setters.

Sesion 5: Interfaces y Polimorfismo

La palabra polimorfismo significa “múltiples formas” y en programación orientada a objetos puede tener varios significados, por ejemplo la habilidad que tiene un método dentro de un objeto con interactuar con diferentes objetos de diferentes clases de la misma forma pero con resultados diferentes. Esto se logra cuando creamos clases que tienen la misma interfaz (es decir los mismos métodos públicos) pero se comportan de manera diferente.

En teoría suena complicado pero con los ejemplos del siguiente video lo aprenderás de manera muy sencilla:

Notas

El polimorfismo es la habilidad de tomar multiples formas: imagina que tienes una Factura y la factura requiere ser impresa como PDF, HTML o como texto plano. Quienes no comprenden estos conceptos suelen hacer esto:

```
<?php


class Invoice {}


class HtmlInvoice extends Invoice {}
```

```
class PdfInvoice extends Invoice {}

class TextInvoice extends Invoice {}
```

Pero ¿Qué sucede si tenemos por ejemplo facturas para pago recurrente, para compra de un solo producto, multiples productos, pagos a contados, pagos a crédito? No podemos seguir abusando de la herencia:

```
<?php

class RecurrentPaymentInvoice extends PdfInvoice {} //Esto ya es abuso de herencia y no tiene sentido
```

Aquí lo mejor es que una factura sea una sola clase:

`class Invoice` y la funcionalidad que pueda cambiar (tomar multiples formas, es decir, polimórfica) se exprese como interfaces que puedan ser: implementadas en clases concretas, instanciadas e inyectadas a la clase, como piezas de lego o de un automóvil que puedas reemplazar:

```
<?php


interface ReportFormat {}

interface InvoiceType {}

interface PaymentType {}
```

Con esto ya es posible crear clases puntuales (para reportes, tipos de pago, tipos de factura, incluso tipo de cliente, etc.) que ya no extenderán de Invoice sino que implementarán una interfaz específica:

```
<?php

class HtmlReportFormat implements ReportFormat
```

Ahora si quieres una factura para pagos recurrentes, a crédito y en HTML ya no tienes que enredarte con una herencia interminable ni repetir código, sino simplemente componerla, por ejemplo de la siguiente forma (o usando el patrón factory que verás más adelante):

```
<?php

$invoice = new Invoice(new RecurrentInvoice(Type::Monthly), new CreditPayment(Carbon::format('3 days')));

$invoice->reportWith(new HtmlReport()); //o: new HtmlReport($invoice);
```

Advertencia: todo esto es un ejemplo inventado que podría adaptarse o no a un caso real, queda de parte de cada desarrollador hacer el análisis correspondiente a cada problema específico.

Diferencia entre clases abstractas e interfaces

La interfaz es sólo un “contrato” que no contiene código. Como has podido ver en el video en la interfaz sólo se define el encabezado de los métodos que deben ser implementados por la clase que implemente dicha interfaz.

Una clase abstracta por el contrario es algo intermedio entre una clase y una interfaz: dado que también contiene un “contracto” en este caso métodos abstractos que deben ser implementados por la clase hija, pero también contiene métodos concretos con código. Más adelante veremos más ejemplos de uso de clases abstractas.

Estas notas fueron agregadas para responder a dudas comunes de nuestros usuarios, si tienes otra pregunta específica de este curso por favor hazla en el canal #php de nuestro Slack:

Sesion 6: Autocarga de clases y nombres de espacio con PHP

Tan importante como aplicar buenas prácticas y patrones de diseño en nuestro código, es organizarlo en archivos de forma coherente. El estándar en PHP es crear un archivo por clases y luego utilizar una función de autocarga (autoload) para cargar dichas clases. Además de ver esto en la lección de hoy, aprenderás qué son los nombres de espacio (namespaces), cómo usarlos y porqué son importantes.

Muchos desarrolladores que escriben aplicaciones orientadas a objetos crean un fichero fuente de PHP para cada definición de clase. Una de las mayores molestias es tener que hacer una larga lista de inclusiones al comienzo de cada script (uno por cada clase).

En PHP 5 esto ya no es necesario. La función spl_autoload_register() registra cualquier número de autocargadores, posibilitando que las clases e interfaces sean cargadas automáticamente si no están definidas actualmente. Al registrar autocargadores, a PHP se le da una última oportunidad de cargar las clases o interfaces antes de que falle por un error.

Sugerencia

Aunque la función __autoload() también puede ser empleada para autocargar clases e interfaces, es preferible utilizar la función spl_autoload_register(). Esto es debido a que es una alternativa más flexible (posibilitando que se pueda especificar cualquier número de autocargadores en la aplicación, tales como los de las bibliotecas de terceros). Por esta razón, se desaconseja el uso de __autoload(), ya que podría estar obsoleta en el futuro.

Nota:
Antes de 5.3.0, las excepciones lanzadas en la función __autoload() no podían ser capturadas en el bloque catch, resultando en un error fatal. Desde 5.3 en adelante, esto es posible simpre que, si se lanza una excepción personalizada, esté disponible la clase de la excepción personalizada. La función __autoload()podría utilizarse recursivamente para cargar la clase de excepción personalizada.

Nota:
La autocarga no está disponible si se utiliza PHP en el modo interactivo CLI.

Nota:
Si el nombre de la clase se utiliza, por ejemplo, en call_user_func(), este puede contener algunos caracteres peligrosos tales como ../. Se recomienda no utilizar la entrada del usuario en tales funciones, o al menos verificar dicha entrada en __autoload().

Ejemplo #1 Ejemplo de autocarga

Este ejemplo intenta cargar las clases *MiClase1* y *MiClase2* desde los ficheros *MiClase1.php* y *MiClase2.php*, respectivamente.

```php
<?php
spl_autoload_register(function ($nombre_clase) {
    include $nombre_clase . '.php';
});

$obj  = new MiClase1();
$obj2 = new MiClase2();
?>
```

Ejemplo #2 Otro ejemplo de autocarga

Este ejemplo intenta cargar la interfaz *IPrueba*.

```php
<?php

spl_autoload_register(function ($nombre) {
    var_dump($nombre);
});

class Foo implements IPrueba {
}

/*
string(7) "IPrueba"

Fatal error: Interface 'IPrueba' not found in ...
*/
?>
```

Ejemplo #3 Autocarga con manejo de excepciones para 5.3.0+

Este ejemplo lanza una excepción y demuestra los bloques try/catch.

```php
<?php
spl_autoload_register(function ($nombre) {
    echo "Intentando cargar $nombre.\n";
    throw new Exception("Imposible cargar $nombre.");
});

try {
    $obj = new ClaseNoCargable();
} catch (Exception $e) {
    echo $e->getMessage(), "\n";
}
?>
```

El resultado del ejemplo sería:

```
Intentando cargar ClaseNoCargable.
Imposible cargar ClaseNoCargable.
```

Cap. 7.- Autocarga de clases con Composer y PSR-4

Hace unos años, cada framework de PHP utilizaba las capacidades de auto-carga de PHP de forma diferente. Esto llevaba a inconsistencias y a que resultara realmente difícil combinar unos componentes o paquetes de PHP con otros. Afortunadamente hoy tenemos la recomendación de estándar PSR-4 y además el manejador de dependencia Composer que también se encarga de autocargar nuestras clases, y esto es lo que veremos en la clase de hoy.

Cap. 8.- Repaso y solución a los ejercicios

En la lección de hoy veremos un repaso de los conceptos de programación orientada a objetos que hemos aprendido hasta ahora como PSR-4, herencia, clases abstractas, inyección de dependencias y refactorización. Además veremos la solución a los principales ejercicios planteados durante la lección 5.

Herencia:
Cuando trabajamos con herencia algunas veces notaremos que un método pertenece a una clase padre o por el contrario no pertenece a la clase padre sino a la clase hija. Una manera sencilla de deducir esto es preguntarte ¿Esta funcionalidad la necesitan todas las clases que heredan de ___ o sólo una clase puntual? Si la respuesta es todas o la mayoría, puedes colocar la funcionalidad en la clase padre, de lo contrario sitúala en la clase hija. Aunque a veces no necesitamos herencia en lo absoluto sino composición.

Inyección de dependencias:
Con la inyección de dependencias podemos "componer" un objeto (en nuestro ejemplo una unidad) para que aún siendo de la misma clase Unit pueda tener características totalmente diferentes (en nuestro ejemplo armas o armaduras distintas). Este concepto lo seguiremos viendo más adelante.

Constructores y setters: podemos pasar dependencias (objetos) u otros valores a un objeto a través de su constructor o a través de métodos setters.

Refactorización:
Es el término que usamos cuando cambiamos código para que mantenga la misma funcionalidad pero esté más ordenado, sea más fácil de entender, mantener y extender.

Métodos VS propiedades:
Recuerda que las propiedades nos dan sólo un valor, mientras que con el método podemos encapsular comportamiento.

Clases abstractas VS Interfaces:
Con ambas podemos forzar que las clases que extiendan de la clase abstracta o implementen de la interfaz cumplan con un "contrato". La diferencia es que las clases abstractas nos permiten agregar funcionalidad extra (porque es una clase padre) y las interfaces sólo representan un contrato.
Algunas veces nos daremos cuenta que necesitamos una en vez de la otra, como veremos en una lección más adelante.

***

En las siguientes lecciones seguiremos evolucionando nuestro juego para seguir otras buenas prácticas y explicarte más patrones de diseño y técnicas de OOP y refactorización.

Cap. 9.-Patrón Factory y Value Objects

En la lección de hoy vamos a aprender sobre un par de patrones de la programación orientada a objetos como lo son **Factory** y **Value Object**. También aprenderemos que muchas veces podemos aplicar refactorización de nuestro código para simplificarlo en vez de hacerlo más complejo.

Patrón Factory

En OOP, factory se refiere a un objeto o método que tiene por objetivo crear o instancias otros objetos.

Esto es una buena práctica y no debe confundirse con el abuso del operador `new` dentro de clases. Más sobre el tema en nuestro curso avanzado de creación de componentes con PHP.

Value Objects

Los value objects nos permiten agrupar grupos de valores que tienen sentido juntos pero no tienen sentido separados. Por ejemplo:

- Dinero: `new Money(50, 'USD'); // cantidad y tipo de moneda`
- Coordenadas: `new Coordinates('38.898648N', '77.037692W'); // latitud y longitud`
- Fecha: `new Date(2016, 07, 21); //año, mes, día`
- Entre muchos otros.

Refactorización

Como habrás notado en esta lección, refactorizar no significa complicar el código, sino adaptarlo a la lógica y requerimientos de nuestra aplicación, y esto implica muchas veces simplificar el código, por ejemplo los cambios en la forma como manejamos las dependencias a través de polimorfismo para las armas y armaduras de nuestro juego de ejemplo hizo innecesaria la herencia en la

clase `Unit` y este tipo de cambios sucede también muy a menudo en aplicaciones reales.

Al refactorizar tu código debes ser muy cuidadoso con no romper la funcionalidad existente de tu aplicación, por ejemplo si estás eliminando una jerarquía asegúrate de reemplazar las subclases en otras partes de tu app y de que realmente no son necesarias, etc. Este tema lo veremos más adelante en uno nuevo curso de refactorización.

***

En la lección siguiente aprenderemos cómo utilizando OOP podemos reducir la cantidad de código estructurado (como condicionales y sentencias switch) de nuestra aplicación.

Cap.10.- Reducción de uso de condicionales IF y sentencias SWITCH

Cuando trabajamos con programación estructurada, tomamos decisiones en nuestro código utilizando condicionales IF o sentencias SWITCH, pero cuando trabajamos con programación orientada a objetos, podemos desarrollar de manera que sean objetos y no decenas de condicionales o switch / case los que tomen las decisiones sobre cómo se debe procesar nuestro programa. Por ejemplo: cuánto descuento se le debe aplicar a un cliente. Esto lo vamos a ver en la lección de hoy con un par de ejemplos sencillos pero muy útiles y fáciles de entender.

Null object

Un null object es una especie de placeholder con el cuál podremos evitar tener que preguntar si un objeto está presente o no, dado que el objeto siempre estará presente así la versión del mismo no cause ningún efecto sobre el programa.

Herencia

Podemos utilizar herencia para colocar los condicionales necesarios dentro de la clase padre y así en las subclases no tendremos más que sobre-escribir los métodos correspondientes. Veremos más sobre esto en una lección siguiente.

***

Con esto finalizamos la parte 1 de este curso, en la segunda parte veremos más características de la OOP en PHP como son métodos y propiedades estáticas, métodos mágicos y más.

Parte 2: Métodos y propiedades estáticas y constantes

Cap.11.- Propiedades y métodos estáticos

Para comenzar la segunda parte de nuestro curso de programación orientada a objetos, te enseñaré no sólo a crear y usar métodos y propiedades estáticos, sino también a cuando es conveniente su uso y cuando no y porqué. También aprenderemos sobre el uso del operador de ámbito `::`.

Propiedades estáticas

Mientras que los valores de las propiedades que hemos visto hasta ahora pertenecen a la instancia de un objeto, una propiedad estática forma parte de la clase:

Quiere decir que todos los objetos que accedan a dicha propiedad estática podrán leer y modificar el mismo valor, y esto la mayoría de las veces no es conveniente y puede traer resultados inesperados y errores más difíciles de depurar.

Incluso en casos donde realmente queremos compartir una información a través de todo el sistema, es mejor utilizar un contenedor de inyección de dependencias, como aprenderás en el curso avanzado de PHP: creación de componentes.

Operador de resolución de ámbito

Para llamar a métodos estáticos, acceder a propiedades estáticas o constantes, usamos el operador de resolución de ámbito `::`. Ejemplos:

- `Str::camelCase('words_with_underscores');` debería retornar: wordsWithUnderscores
- `Unit::MAX_DAMAGE`
- `Diccionario::$words`

Métodos estáticos y código acoplado

Cuando llamamos a un método estático dentro de un método de otra clase, estamos atando una clase a otra. En el ejemplo de la lección, ahora nuestra clase Attack requiere obligatoriamente de la clase `Styde\Translator` debido al llamado a `Translator::get()` mientras que `Armor` y `Weapon` son reemplazables dentro de la clase `Unit` puesto que se pasan por inyección de dependencias. Verás más sobre este tema en nuestro curso avanzado de PHP: creación de componentes.

Named constructors

Los métodos estáticos son bastante útiles para crear los llamados named constructors, como por ejemplo:

- `Unit::createSoldier('Silence')` es el equivalente a `new Soldier('Silence', new BasicSword());`
- `Carbon::now()` es el equivalente a `new Carbon();`

***

En las siguientes 2 lecciones aprenderás sobre buenas prácticas y trucos en el uso de métodos estáticos.

Cap.12.- Constructores semánticos e interfaces fluidas

A pesar de todo, los métodos estáticos tienen una serie de utilidades interesantes y hoy te voy a explicar sobre una de ellas, que en inglés se denomina “named constructors” y nosotros podríamos traducir por constructores semánticos, además veremos sobre los “fluent interfaces” o interfaces fluidas que son bastante usadas dentro del framework Laravel.

Named constructors

Para crear un constructor semántico sólo tienes que crear un método estático que cree y retorne una nueva instancia de una clase:

```
public static function createSoldier($name)
{
    return new Unit($name, new BasicSword)
}
```

Aunque en el video no pasé el parámetro $name a través del método, fíjate cómo es perfectamente posible. Así que te invito a hacer este cambio y a crear otro método para otro tipo de unidad.

```
public static function newAdmin(array $attributes)
{
    $admin = new User($attributes);
    $admin->role = 'admin'

    return $admin;
}
```

Como aprendiste en una lección anterior, un named constructor es a su vez un método factory.

Fluent interfaces

Para crear interfaces fluidas sólo tienes que recordar retornar `$this` luego de cada función. Las interfaces fluidas suelen usarse para métodos que modifican el estado de un objeto (como un setter).

```
1 function setArmor(Armor $armor)
2 {
3     $this->armor = $armor;
4
5     return $this;
6 }
```

```
1 class Redirect
2 {
3     public function withErrors(errors)
4     {
5         $this->errors = $errors;
6 
7         return $this;
8     }
9 }
```

En la siguiente lección aprenderemos sobre el uso de métodos estáticos en la creación de “facades”.

Cap.13.- Qué son los facades y cómo implementarlos en tu proyecto

El uso de métodos estáticos es muy sencillo puesto que nos permite invocar a un método en cualquier lugar de nuestro sistema, sin tener que preocuparnos por inyectar dependencias y ni siquiera por crear una nueva instancia de una clase. Pero esta facilidad de uso viene con un costo: terminamos con un sistema menos flexible y más acoplado. Aquí es donde entra el concepto de facades en PHP, ideado por Taylor Otwell para Laravel 4, las cuáles son el punto intermedio entre una buena arquitectura y una interfaz fácil de usar como aprenderás en esta lección.

Notas
Por Clemir Rondón.

- Los principios SOLID, acrónimo de Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion, representan cinco principios básicos de la programación orientada a objetos.
- En caso que quieras eliminar el archivo helpers.php recuerda eliminarlo también del archivo composer.json y, luego ejecuta en consola:

```
1 composer dump-autoload
```

- Para guardar el contenido del mensaje del Log en un archivo usa el método file_put_contents de PHP con la bandera `FILE_APPEND` para en caso que ya el archivo exista, añada la información en él, en vez de sobrescribirlo.
- El principio SOLID: Open-closed (principio abierto/cerrado) establece que las clases u otra entidad de software deben estar abiertas para su extensión y cerradas para su modificación.
- El principio SOLID Dependency inversion (principio de inversión de la dependencia) nos indica que debemos depender de abstracciones en vez de implementaciones concretas.
- En el curso Curso de creación de componentes para PHP y Laravel puedes estudiar cómo el framework Laravel implementa la inyección de dependencias, facades y muchos otros temas que te ayudarán a entender cómo funciona Laravel por dentro.

Actividad

Implementa el concepto de facade para la clase translator que se desarrolló en la lección Métodos y propiedades estáticos.

***

En la siguiente lección aprenderemos sobre el uso de constantes en PHP.

Cap.14.- Cómo declarar y usar constantes de clases

En esta lección aprenderás a declarar constantes dentro de tus clases y a usarlas en cualquier lugar de tu aplicación. Además hablaremos del anti-patrón de números mágicos, y algunos adelantos de la versión 7.1 de PHP.

Notas

Pronto agregaremos notas a esta lección.

***

En la siguiente parte de este curso aprenderemos sobre el uso de métodos mágicos en PHP y algunos trucos usados por el ORM Eloquent de Laravel.

Cap.15.- Uso de los métodos mágicos __get, __set, __isset y __unset con PHP

PHP es un lenguaje muy dinámico y ofrece a los desarrolladores la posibilidad de declarar cualquier lógica para acceder a propiedades o llamar a métodos dentro de una clase, aunque estos no estén definidos previamente. Esto es lo que se conoce como métodos mágicos y algunos componentes de PHP como por ejemplo el ORM Eloquent de Laravel, hacen uso de esta característica para mostrarnos una API más dinámica y fácil de usar, y por supuesto este tema lo veremos a partir de hoy en nuestro curso de programación orientada a objetos con PHP.

Notas

Por Clemir Rondón.

- Puedes trabajar con las propiedades dinámicamente, por ejemplo, si se llama a `$this->$key = $value;` PHP está considerando a $key como una variable que será evaluada antes de asignarle un valor.
- Los métodos mágicos en PHP son aquellos métodos especiales que se llaman automáticamente. Estos comienzan con el símbolo (__) seguido de una palabra reservada tales como:__construct, __get, __set, __isset, etc. Estos debe ser declarados y definidos por el desarrollador dentro de la clase y no serán

llamados explícitamente sino que se invocarán según el evento que ocurra.

- **__get()** es el método mágico con el cual podemos obtener el valor de una propiedad de una clase dinámicamente, cuando ésta es inaccesible (la propiedad está declarada como private o protected) o no existe en la clase.

public __get (string $name)

- **__set()** es el método mágico que PHP llamará automáticamente (si está declarada en la clase) cuando queremos definir una propiedad inaccesible o inexistente. Este método recibe dos argumentos: el nombre de la propiedad que intentas acceder y el valor que le quieres asignar.

  public __set (string $name, mixed $value)