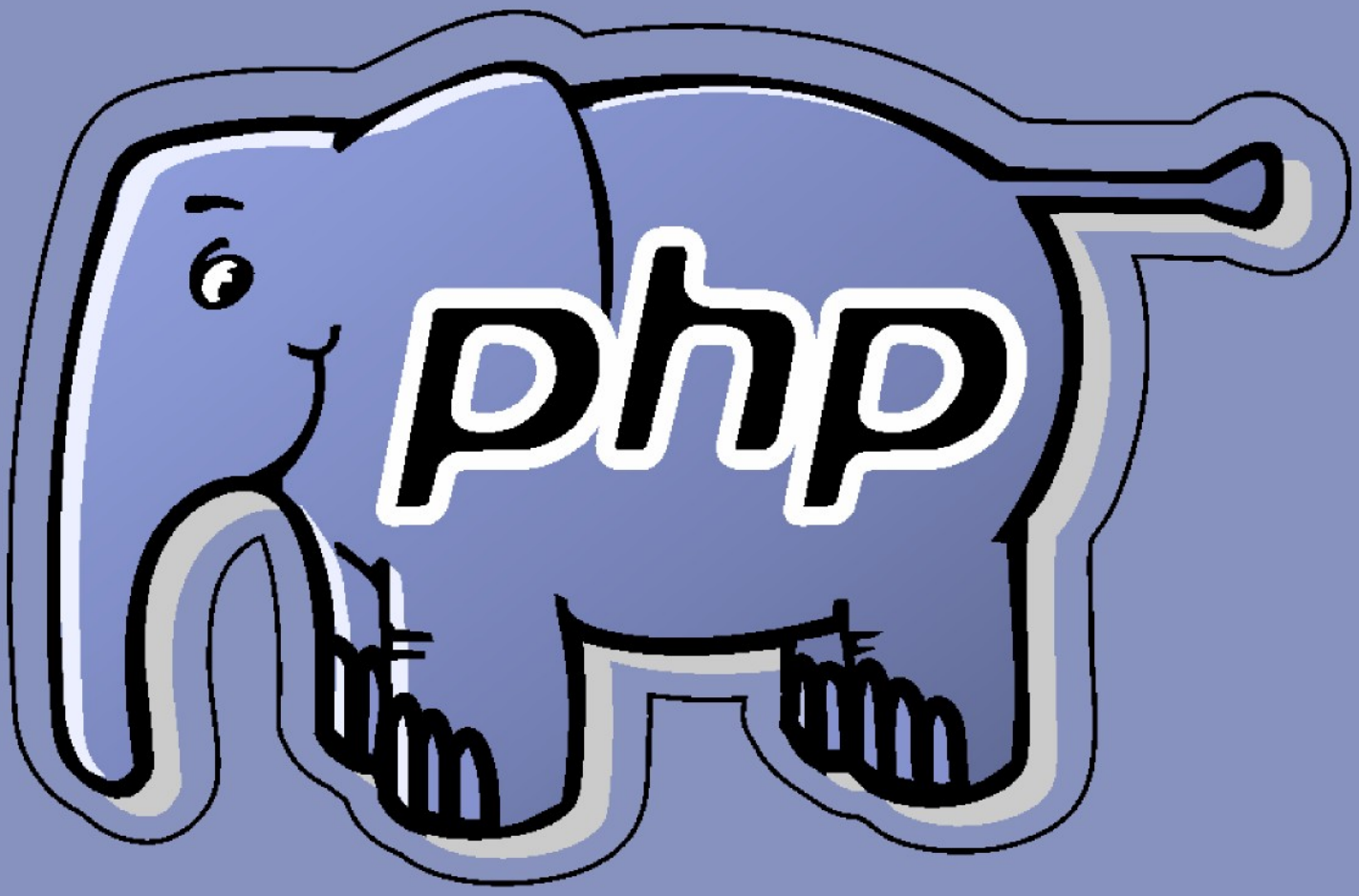
Programación en



Versión 2.0

php

Gugler
Laboratorio de Investigación Gugler

FCyT
Facultad de Ciencia y Tecnología

UADER
Universidad Autónoma de Entre Ríos

CONTENIDO

Capítulo 5

POO en PHP - Patrones

Introducción

Finalizando los temas referidos a POO haremos un pequeño repaso mediante un ejemplo de una aplicación diseñada orientada a objeto en la cual generaremos una página html con encabezado, cuerpo y pie. En la próxima clase nos centraremos en el estudio de algunos patrones de diseño y su utilización.

Repaso POO

Caso de estudio

Mediante este ejemplo crearemos, haciendo uso de los conceptos de POO vistos hasta el momento, una estructura aplicable a un sitio web, la cual constará de un *encabezado,* un *menú*, el *cuerpo* y el *pié* a página.

Diagrama UML

Se adjunta en otro archivo y como anexo el diagrama UML del esquema a estudiar.

Scripts

Para la realización del ejemplo desarrollaremos los siguientes script:

- index.php
- Pagina.php
- Cabecera.php
- Menu.php
- Opcion.php
- Pie.php

Codificación de los script

Index.php

```
require_once 'Pagina.php';
require_once 'Cabecera.php';
require_once 'Pie.php';
require_once 'Menu.php';
require_once 'Opcion.php';

class Index
{
	public function ejecutar()
	{


	// Se crea la cabecera de la Página con nombre y descripción
		$oCabecera = new Cabecera('Página Prueba', 'descripción de página de prueba');

	// Se crea el menú de la página
		$oMenu = new Menu('Principal', 'Menú principal', 'horizontal');
```

```
        // Se crean las opciones de menú para la página
                $oOpcionInicio = new Opcion('Inicio', 'Página de Inicio', 'index.php');
                        $oMenu->agregarOpcion($oOpcionInicio);
                $oOpcionProductos = new Opcion('Productos', 'Página de Productos', 'productos.php');
                        $oMenu->agregarOpcion($oOpcionProductos);
                $oOpcionServicios = new Opcion('Servicios', 'Página de Servicios', 'servicios.php');
                        $oMenu->agregarOpcion($oOpcionServicios);
                $oOpcionEnlaces = new Opcion('Enlaces', 'Página de Enlaces', 'enlaces.php');
                        $oMenu->agregarOpcion($oOpcionEnlaces);
                $oOpcionContacto = new Opcion('Contacto', 'Página de Contacto', 'contacto.php');
                        $oMenu->agregarOpcion($oOpcionContacto);

        // Se crea el pie de página
                $oPie = new Pie('2010', '2010','GUGLER', 'webmaster@gugler.com.ar',
                        'http://www.gugler.com.ar', 'Todos los derechos resevados.');


        // Se crea el objeto de clase Pagina con un título
                $oPagina = new Pagina('Prueba');

        // Se agregan los objetos a la página
                $oPagina->agregarCabecera($oCabecera);
                $oPagina->agregarMenu($oMenu);
                $oPagina->agregarPie($oPie);

        // Se muestra el código HTML
                echo $oPagina->dibujarHtml();

        }
}

$oIndex = new Index();
$oIndex->ejecutar();
```

Como podemos ver, en *index.php* definimos una clase *Index* en la cual generamos todos los objetos necesarios para la creación de nuestra página dentro del método *ejecutar*. Finalmente instanciamos un objeto e invocamos a dicho método.

Analizando el método *ejecutar*, vemos que comenzamos creando los distintos objetos que componen nuestra página.

Para el objeto cabecera vemos (al analizar la clase Cabecera) que su constructor espera como parámetro el nombre y la descripción de nuestro sitio. Estos parámetros son asignados a dos atributos privados de la clase (*$_nombreWeb y $_descripcion*) y luego, en base a estos valores, se genera el código *html* necesario para escribir el encabezado en nuestra página. Esto se realiza en el método *generarHTML().*

Cabecera.php

```
class Cabecera
{
	private $_nombreWeb;
	private $_descripcion;

	public function __construct($nombre, $descripcion)
	{
		$this->_nombreWeb = $nombre;
		$this->_descripcion = $descripcion;
	}

	public function generarHtml()
	{
		$cadena = '';
		$cadena .= '<h1>'.$this->_nombreWeb.'</h1>';
		$cadena .= '<br/>';
		$cadena .= '<h3>'.$this->_descripcion.'</h3>';
		$cadena .= '<br/>';
		$cadena .= '<hr/>';
		return $cadena;
	}
}
```

El segundo objeto a crear es el *Menu*. De la misma forma que ocurría con el objeto cabecera, podemos ver que su constructor espera tres parámetro los cuáles son asignados a atributos privados de la clase.

Como una particularidad de esta clase podemos ver que uno de sus atributos es un array *($_opcion).* La idea es que este arreglo guarde todas las opciones que el menú deba presentar en el sitio. Cada una de estas opciones será un objeto cuya estructura está definida en la clase Opcion. Esta clase contiene ciertos atributos los cuáles deberán ser establecidos a la hora de crear una opción y un método que será el encargado de crear el *html* a incrustar en el menú.

El ejemplo en si busca crear tantos objetos *Opcion* como opciones tenga el menú y agregarlos por medio del método público *agregarOpcion(Opcion $opcion)* al arreglo $_opciones de la clase *Menu*.
Una vez agregadas todas las opciones, llamamos al método *generarHTML() (clase Menu)* el cual recorrerá todo nuestro arreglo y escribirá *(invocando al método de cada objeto Opcion guardado en el arreglo)* el código *html* del menú.

Menu.php

```
require_once 'Opcion.php';
```

```
class Menu
{
	private $_nombreMenu;
	private $_ubicacion;
	private $_descripcion;
	private $_opcion = array();

	public function __construct($nombeMenu, $ubicacion='horizontal', $descripcion)
	{
		$this->_nombreMenu = $nombreMenu;
		$this->_ubicacion = $ubicacion;
		$this->_descripcion = $descripcion;
	}

	public function agregarOpcion(Opcion $opcion)
	{
		$this->_opcion[] = $opcion;
	}

	public function generarHtml()
	{
		$cadena = '';
		foreach ($this->_opcion as $oOpcion) {
			$cadena .= $oOpcion->generarHtml().' ';
		}
		$cadena .= '<br/><hr/>';
		return $cadena;
	}
}
```

Opcion.php

```
class Opcion
{
	private $_nombreOpcion;
	private $_descripcion;
	private $_enlace;

	public function __construct($nombreOpcion, $descripcion, $enlace)
	{
		$this->_nombreOpcion = $nombreOpcion;
		$this->_descripcion = $descripcion;
		$this->_enlace = $enlace;
	}

	public function generarHtml()
	{
		$cadena = '';
```

```
                $cadena .= "<a href='".$this->_enlace."' title='".$this->_descripcion."'>".$this-
                        >_nombreOpcion."</a>";

                return $cadena;
        }
}
```

El próximo objeto a crear es el *pie*. El mismo requerira algunos valores como atributos y nuevamente dispondrá de un método *generarHTML()* el cual invocaremos para escribirlo en nuestra página.

Pie.php

```
class Pie
{
        private $_anioDesde;
        private $_anioHasta;
        private $_nombreAutor;
        private $_emailAutor;
        private $_webAutor;
        private $_descripcion;

        public function __construct($anioDesde, $anioHasta, $nombreAutor, $emailAutor, $webAutor,
$descripcion)
        {
                $this->_anioDesde = $anioDesde;
                $this->_anioHasta = $anioHasta;
                $this->_nombreAutor = $nombreAutor;
                $this->_emailAutor = $emailAutor;
                $this->_webAutor = $webAutor;
                $this->_descripcion = $descripcion;
        }

        public function generarHtml()
        {
                $cadena = '';
                $cadena .= '<hr/>';
                $cadena .= '<p>&copy;'.$this->_anioDesde.'-'.$this->_anioHasta;
                $cadena .= ' '.$this->_nombreAutor.' - '.$this->_descripcion.'<br/>';
                $cadena .= $this->_emailAutor.' - '.$this->_webAutor.'<br/>';

                return $cadena;
        }
}
```

Como último objeto tenemos a *Pagina.* Su clase define tres atributos privados (titulo, cabecera, menu y pie). Esta es el objeto en sí que unirá a todos los otros objetos que fuimos creando.
El método agregarCabecera() esperará como parámetro un objeto de estructura Cabecera. El

método *agregarPie()* esperará objeto de estructura *Pie* y de lo mismo ocurrirá con *agregarMenu()*.

Pagina.php

```
require_once 'Cabecera.php';
require_once 'Pie.php';
require_once 'Menu.php';

class Pagina
{
	private $_titulo;
	private $_cabecera;
	private $_pie;
	private $_menu;

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

	public function agregarCabecera(Cabecera $cabecera)
	{
		$this->_cabecera = $cabecera;
	}

	public function agregarPie(Pie $pie)
	{
		$this->_pie = $pie;
	}

	public function agregarMenu(Menu $menu)
	{
		$this->_menu = $menu;
	}

	public function dibujarHtml()
	{
		$cadena = '';
		$cadena .= '<html><head><title>'.$this->_titulo.'</title></head>';
		$cadena .= '<body>';
		$cadena .= $this->_cabecera->generarHtml();
		$cadena .= $this->_menu->generarHtml();
		$cadena .= $this->_pie->generarHtml();
		$cadena .= '</body>';
		$cadena .= '</html>';

		return $cadena;
```

```
        }
}
```

El método encargado de generar nuestra página *html* es *dibujarHtml().* Si lo analizamos un poco, vemos que se generará una secuencia de etiquetas html las cuales se irán guardando en la variable $cadena. Agregaremos también el código *html* generado por cada método *generarHtml()* de los objeto que fuimos agregando (de los objetos Cabecera, Menu y Pie). Debemos poner atención ya que al invocar, por ejemplo *$this->_cabecera->generarHtml()*, estamos invocando un método definido en la clase Cabecera, pero que forma parte del objeto que nosotros pasamos como parámetro a nuestra clase. Finalmente agregamos los tag de cierre y devolvemos el código para ser mostrado luego mediante una instrucción *echo*.

Herencia

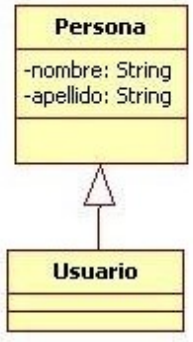
Definición

Es una característica que nos provee la POO mediante la cual creamos una clase que va a heredar los métodos y atributos de otra ya definida, por lo que podemos decir que la clase que hemos creado es una subclase.

Es una relación de parentesco entre dos entidades (una es padre de la otra, una es hija de la otra), en la cual los objetos hijos heredan las propiedades y el comportamiento de todas las clases que son sus padres. Uno de los peores errores que se puede cometer , es usar el mecanismo de la herencia con el único objetivo de *reusar código* de otra clase.

Representación UML

La herencia en UML es representada mediante una *flecha continua* que va desde la clase *hija* hasta la clase *padre.* Ejemplo:



¿Cómo se heredan los atributos y métodos?

La herencia de los métodos y atributos se ajusta a las siguientes reglas:

1. Todos los atributos y métodos se heredan del Padre al Hijo, pero…
2. Los atributos y métodos que son de tipo *público (public)* o *protegido (protected)* son visibles *directamente* desde el Hijo (en el caso de *público* son visibles de todos lados, incluyendo desde el exterior del objeto).
3. Los atributos y métodos que son de tipo *privado (private)* se heredan del padre a su hijo pero *no son visibles* de forma directa, solo se podrá hacer a través de métodos del padre que sean públicos o protegidos.

Nota: podríamos decir que un método público *saludar* del padre que su hijo hereda se ejecuta de la misma forma que si el hijo lo hubiera implementado él mismo, solo que el código se encuentra *almacenado* en el padre.

¿Cómo se codifica?

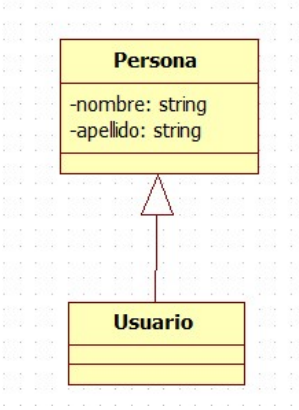
Para que una clase sea heredada de otra clase ya creada deberemos usar la palabra reservada *extends.*

```
class Persona
{
        private $_nombre;
        private $_apellido;
 }

class Usuario extends Persona
{
}
```

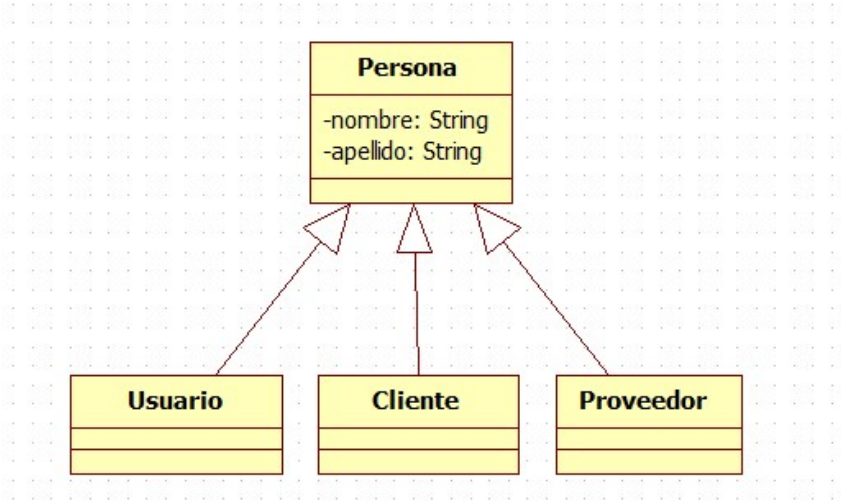
Especialización

Una herencia por especialización es la que se realiza cuando necesitamos crear una clase nueva que disponga de las mismas características que otra pero que le añada funcionalidades. Por ejemplo, imaginemos que solo contamos con la clase Persona y descubrimos que necesitamos una clase Usuario, podemos decir entonces que a partir de Persona hicimos una *especialización* al crear la clase Usuario, ya que tomamos un comportamiento más genérico y lo adecuamos a una clase mucho más concreta y aplicable a entornos más particulares.



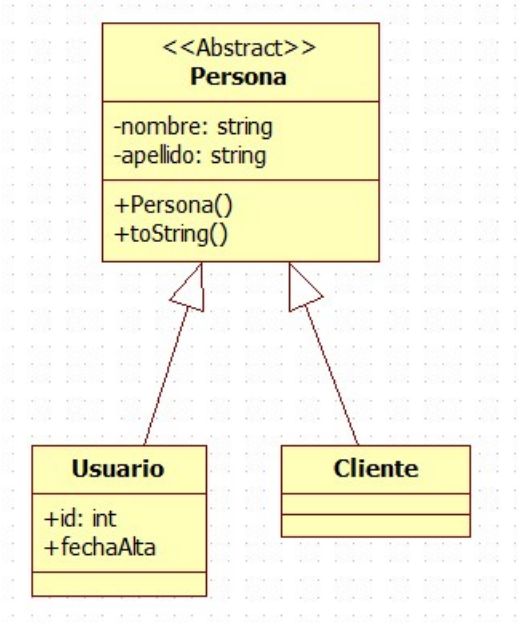
Generalización

La herencia por generalización es la que realizamos cuando tenemos muchas clases que comparten unas mismas funcionalidades y por homogeneizar las partes comunes se decide crear una clase que implemente toda esa parte común y se dejan solo las partes especificas en cada clase.



Clases Abstractas

Es un tipo especial de clase que no puede ser instanciada directamente pero que sirve de *esqueleto* para la declaración de otras clases.



En UML se representa con el nombre de la clase en cursiva ó agregando un *estereotipo* (texto que va arriba del nombre de la clase entre <<>>).

¿Cómo se heredan los atributos y métodos?

La herencia de los métodos y atributos se ajusta a las mismas reglas que la herencia:

1. Todos los atributos y métodos se heredan del Padre al Hijo, pero…
2. Los atributos y métodos que son de tipo *público (public)* o *protegido (protected)* son visibles *directamente* desde el Hijo (en el caso de *público* son visibles de todos lados, incluyendo desde el exterior del objeto).
3. Los atributos y métodos que son de tipo *privado (private)* se heredan del padre a su hijo pero *no son visibles* de forma directa, solo se podrá hacer a través de métodos del padre que sean públicos o protegidos.

¿Cómo se codifica?

En este tipo de clases debemos indicar que la misma no puede ser instanciada. Esto lo

hacemos anteponiendo la palabra abstract a la palabra reservada class. Representaremos ahora este ejemplo en PHP:

```
abstract class Persona
{
        private $_nombre;
        private $_apellido;
 }

class Usuario  extends Persona
{
        //definición del resto de la clase
}

class Cliente extends Persona
{
        //definición del resto de la clase
}
```

Lo que estamos diciendo es que la clase no puede ser usada directamente y que nos servirá de molde para crear otras clases que sí serán concretas y candidatas a instanciar.

```
$persona = new Persona();                    //no será factible de realizar
```

Sobre-escritura de métodos

Es la característica de la POO que nos permite la re-definición de los métodos de la clase base en las subclases, esto es, poder sobre escribir un método que heredamos de nuestro padre. Esto se puede hacer simplemente volviendo a definir en la case *hija* un método con el mismo nombre.

```
class ClasePadre
{

        protected function miFuncion()
        {
                echo ' Metodo Padre ';
        }
}

class Extendida extends ClasePadre
{
```

```
        public function miFuncion()
        {
                parent::miFuncion;
                echo ' Metodo hijo';
        }
}
```

Polimorfismo

En POO el polimorfismo es la capacidad de poder tomar varias clases que tiene operaciones iguales y poder invocar a cualquier instancia el mismo método común. Esto nos permite poder hacer un diseño genérico que se aplique a varios objetos que tienen operaciones comunes.

Para entender el concepto, analizaremos el siguiente ejemplo:

Tenemos una clase Impresora que se encarga de recibir cualquier cosa y la imprime con un *echo*.

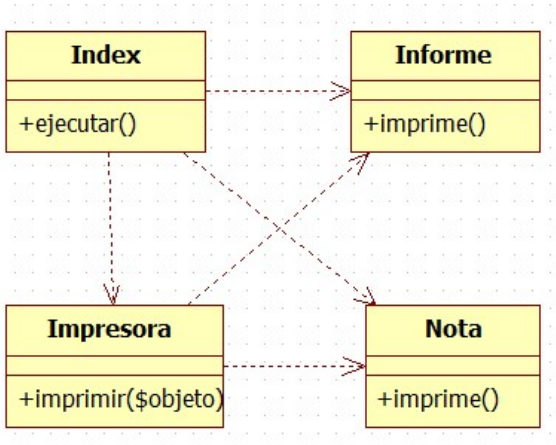
```
class Impresora
{
        public function imprimir($algo)
        {
                echo $algo;
        }
}

 $impresora = new Impresora();
$impresora->imprimir('Hola mundo!);
```

La idea detrás de éste modelo es que deberíamos poder recibir objetos que ellos mismos sepan cómo imprimirse, y que la impresora solamente se encargue de la acción (del echo).

Para esto definimos que cada objeto que quiera imprimirse deberá tener un método *imprime* y además deberá saber qué información debe entregar a la impresora.

Con esta información definimos el siguiente diagrama de clases:



Codificación

index.php

```
require_once ('Impresora.php');
require_once ('Informe.php');
require_once ('Curriculum.php');

Impresora::imprimir(new Informe());
Impresora::imprimir(new Curriculum());
```

Impresora.php

```
abstract class Impresora
{
	public function imprimir($algo)
	{
		echo $algo->imprime();
	}
}
```

Informe.php

```
class Informe
{
	public function imprime()
	{
		return 'imprimo informe';
	}
 }
```

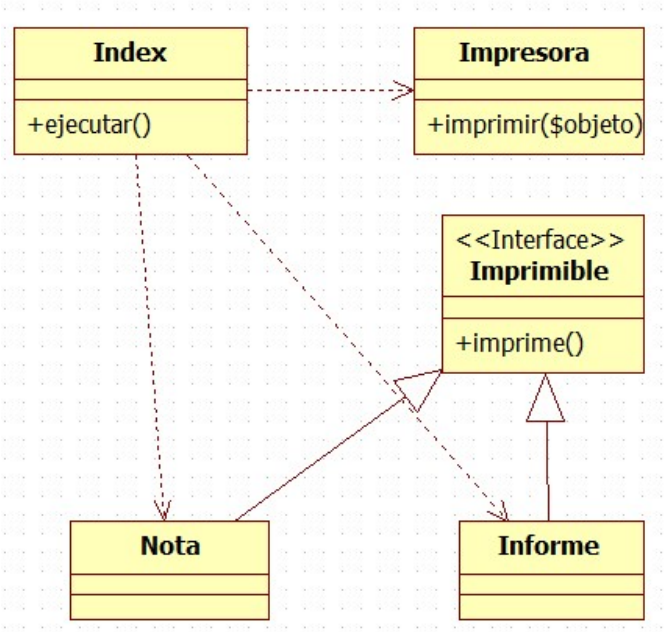
Del código podemos ver que nuestra clase de impresión recibe distintos tipos de objetos que comparten el mismo nombre común de un método que se requiere para que la impresora pueda funcionar. En distintas etapas de nuestro sistema veremos distintas instancias haciendo uso del mismo código de impresión: *polimorfismo*, *muchas formas*.

Interfaces

Una interface es una clase especial en la que solamente se definen los métodos y propiedades *que una clase que la implemente debe codificar*. Las interfaces representan un contrato, de forma que cualquier clase que la implemente debe utilizar los miembros de la interfaz usando la misma forma en que ésta la ha descrito: mismo número de argumentos, mismo tipo de datos devuelto, etc.

Interfaces en UML

Las interfaces, tal como las clases abstractas. no se pueden instanciar. Sus métodos deben ser re-escritos por la clase que implemente.



Gracias a la implementación de interfaces podemos crear relaciones entre clases que no estén derivadas de la misma clase base, pero que tengan métodos comunes, al menos en la forma, aunque no necesariamente en el fondo.

Codificación

ImprimibleInterfaz.php

```
interface Imprimible
{
	public function imprime();
}
```

Informe.php

```
class Informe implements Imprimible Interface
{
	public function imprime()
	{
		return 'Se imprime informe';
	}
}
```

Impresora.php

```
class Impresora
{
	public function imprimir(Imprimible $objeto)
	{
		echo $objeto->imprime();
	}
}
```

index.php

```
require_once('Impresora.php');
require_once('Curriculum.php');
require_once('Informe.php');

abstract class Index
{
	public function run()
	{
		$impresora = new Impresora();
		$impresora ->imprimir(new Curriculum());
		$impresora ->imprimir(new Informe());
	}
}
Index::run()
```

Inmediatamente que agregamos el tipo Imprimible en el método de la Impresora, cualquier

objeto que quiera pasar por ahí deberá implementar la interfaces Imprimible. Una vez que lo implemente, el compilador de PHP le dirá que su clase no tienen el método *imprime* (esto lo obliga la interfaces, usted no la está cumpliendo), por lo tanto para que pueda ser aceptada tiene que contar con ese método.

La palabra reservada final

La palabra clave final aparece con la versión 5 de PHP, que impide que las clases hijas sobrescriban un método, antecediendo su definición con la palabra final.

Si la propia clase se define como final, entonces no se podrá heredar de ella.

Las propiedades no pueden declararse como final. Sólo pueden las clases y los métodos.

Ejemplo de uso de la palabra reservada final en la definición de un método de un clase

```
<?php
        class Base
        {
                public function prueba() {
                   echo "llamada a Base::prueba()\n";
                }
                final public function otraPrueba() {
                   echo "llamada a Base::otraPrueba()\n";
                }
        }

        class Hija extends Base
         {
                public function otraPrueba() {
                   echo "llamada a Hija::otraPrueba()\n";
                }
}
// Devuelve: Cannot override final method Base::otraPrueba()
```

Ejemplo de uso de la palabra clave final en la definición de la clase

```
<?php
final class Base
{
          public function prueba() {
```

```
            echo "llamada a Base::prueba()\n";
        }
        // Aquí no importa si definimos una función como final o no
        final public function otraPrueba() {
            echo "llamada a Base::otraPrueba()\n";
        }
}

class Hija extends Base {
}
// Devuelve un error Fatal: Class Hija may not inherit from final class (Base)
```

La palabra reservada static

Declarar propiedades o métodos de clases como estáticos los hacen accesibles sin necesidad de una instanciación de la clase. Una propiedad declarada como static no puede ser accedida con un objeto de clase instanciado (pero si se puede con métodos estáticos).

Debido a que los métodos estáticos se pueden invocar sin tener creada una instancia del objeto, la pseudo-variable $this no está disponible dentro de los métodos declarados como static.

Las propiedades estáticas no pueden ser accedidas a través del objeto utilizando el operador flecha (->).

Ejemplo de uso de la palabra reservada static en la definición de atributos de una clase

Prueba.php

```
<?php

class A
{
    public static $my_static = 'A';

    public function staticValue() {
        return self::$my_static;
    }

}

class B extends A
{
    public function aStatic() {
```

```
        return parent::$my_static;
    }

}

print A::$my_static . "\n";

$oA = new A();
print $oA->staticValue() . "\n";
print $oA->my_static . "\n";      // Undefined "Property" my_static

print $oA::$my_static . "\n";
$classname = 'A';
print $classname::$my_static . "\n"; // A partir de PHP 5.3.0

print B::$my_static . "\n";
$oB = new B();
print $oB->aStatic() . "\n";
```

Ejemplo de uso de la palabra reservada static en la definición de métodos de una clase

Prueba.php

```
<?php

class Prueba {
    public static function unMetodoEstatico() {
        // ...
    }
}

Prueba::unMetodoEstatico();

$classname = 'Preuba';
$classname::unMetodoEstatico(); // A partir de PHP 5.3.0
```

Un método estático es un método que pertenece a una clase pero no puede acceder a los atributos de una instancia de la misma (objeto). La característica fundamental es que un método estático se puede llamar sin tener que crear un objeto de dicha clase. Un método estático es lo más parecido a una función de un lenguaje estructurado, solo que se lo encapsula dentro de una clase.

Ejemplo:

```
<?php
	class Cadena
	{
		 public static function largo($cad)
		{
			return strlen($cad);
		}
		public static function mayusculas($cad)
		{
			return strtoupper($cad);
		}
		public static function minusculas($cad)
		{
			return strtolower($cad);
		}
	}
	$c='Hola a todos';
	echo 'Cadena original:'.$c;
	echo 'Largo:'.Cadena::largo($c);
	echo 'Toda en mayúsculas:'.Cadena::mayusculas($c);
	echo 'Toda en minúsculas:'.Cadena::minusculas($c);
?>
```

En el ejemplo podemos ver la forma de definir un método estático, el cuál comienza en el modificador de acceso y la palabra reservada *static*.

```
 public static function largo($cad)
 {
	return strlen($cad);
}
```

Además, vemos que podremos acceder a dicha función sin la necesidad de instanciar un objeto.

```
	echo 'Largo:'.Cadena::largo($c);
```