

# Lenguajes Interpretados en el Servidor

Objetos en PHP  
(Segunda parte)

# Sobrecarga

- La sobrecarga de propiedades y métodos en PHP se lleva a cabo por medio de los métodos especiales *`__get()`*, *`__set()`* y *`__call()`*.
- Los métodos *`__get()`* y *`__set()`* son invocados de forma automática cuando se intenta acceder a una propiedad o método inexistente en el objeto.
- *`__set()`*, permite establecer un valor para una propiedad, *`__get()`*, permite recuperar u obtener el valor de la propiedad, mientras que *`__call()`*, permite, además del acceso a métodos no definidos en el objeto, implementar la sobrecarga de métodos.

# Sobrecarga de miembros

- Los miembros de una clase pueden ser sobrecargados utilizando los métodos ***\_\_set()*** y ***\_\_get()***.
- La sintaxis de estos métodos es la siguiente:

```
void __set(string $name, mixed $value);  
mixed __get(string $name);
```

En donde, ***\$name*** es el nombre de la propiedad que debe ser asignada (set) u obtenida (get). ***\$value*** es el valor que se intenta establecer (set) al miembro no existente del objeto.

- Los métodos ***\_\_set()*** y ***\_\_get()*** son llamados únicamente si la propiedad referenciada no existe en el objeto. El uso del método ***\_\_get()***, supone una ejecución previa de ***\_\_set()*** para poder acceder a una propiedad que no ha sido definida en el objeto.

# Sobrecarga de métodos

- Para implementar la **sobrecarga** de métodos se utiliza el método **`__call()`**, diseñado para tener acceso a métodos no definidos para el objeto.
- La sintaxis de este método es la siguiente:

```
mixed __call(string $name, array $arguments);
```

En donde, **`$name`**, es el nombre del método invocado que no está definido, y **`$arguments`** contiene los argumentos que serán pasados al método en una matriz.

- Utilizar **`__call()`** para implementar la **sobrecarga** es viable en PHP, sin embargo, debe recordar que en PHP los argumentos pueden tomar cualquier tipo de dato válido y que, además, PHP soporta lista de argumentos de longitud variable. Por tal razón, puede que no sea tan útil esta función.

# Ejemplo de sobrecarga

```
class sinPropiedades {  
    function __set($propiedad, $valor){  
        echo "Asignamos $valor a $propiedad";  
        $this->propiedad = $valor;  
    }  
    function __get($propiedad){  
        echo "Acceso a la propiedad $propiedad(clase ",  
__CLASS__,")\n";  
        return $this->propiedad;  
    }  
    function __call($metodo, $parametros){  
        echo "Acceso al método $metodo (clase", __CLASS__,  
")\nArgumentos:\n", var_dump($parametros), "\n";  
    }  
} //Fin clase sinPropiedades
```

# Ejemplo de sobrecarga

```
//Creación de un nuevo objeto sinPropiedades
$obj = new sinPropiedades();

//Asignando valores a dos propiedades no definidas
$obj->nombre = "Sergio";
$obj->edad = 25;

//Hacer un volcado del objeto
echo var_dump($obj), "\n";

//Acceder a las propiedades sobrecargadas y a otra inexistente
echo 'Nombre: ', $obj->nombre, "\n";
echo 'Edad: ', $obj->edad, "\n";
echo 'Apellido: ', @$obj->apellido, "\n";

//Intentar ejecutar un método inexistente
echo $obj->darNombre('Sergio', 'Pérez', 30);
```

# Herencia de clases (derivación)

- El concepto de herencia en POO es la relación que se da entre dos clases por la cual la clase denominada hija, subclase o derivada, además de contar con sus propias propiedades y métodos, tiene a disposición, por herencia, los miembros definidos en la clase denominada clase padre o superclase.



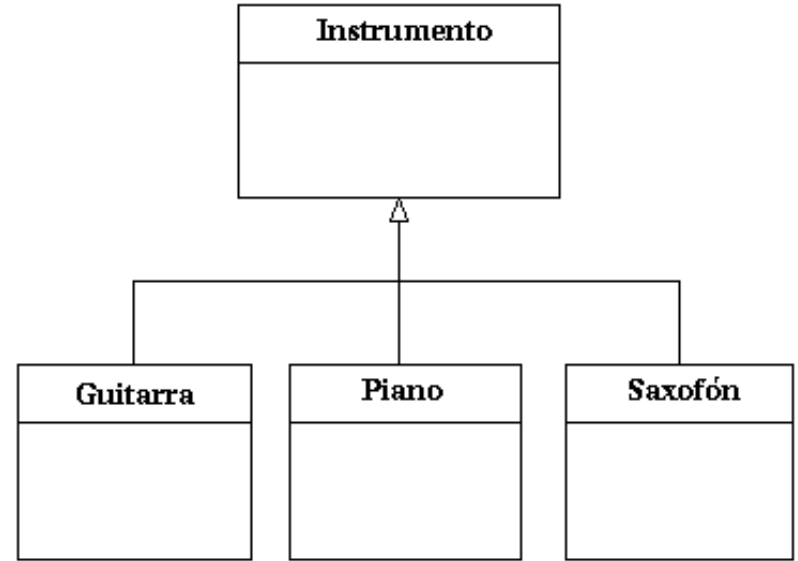
# Herencia de clases (derivación)

- Los lenguajes de programación orientados a objetos permiten definir clases en términos de otras clases y PHP no es la excepción.
- De modo que si se define una **clase persona** con propiedades evidentes como el nombre, sexo, edad, estado civil, estatura, peso, etc. Y métodos como despertar, comer, caminar, dormir, etc, es factible definir **clases derivadas o subclases** como **empleado, estudiante, trabajador** a partir de aquélla.
- Por otra parte, la **clase persona** es **clase padre o subclase** de las **clases empleado, estudiante y trabajador**.



# Herencia de clases (derivación)

- El propósito principal de la herencia es la reutilización del código, definiendo clases en términos de otras clases.
- Una clase hija puede volver a definir uno, alguno o todos los métodos, propiedades y constantes de la superclase para proporcionar una funcionalidad adicional o diferente.



# Herencia de clases (derivación)

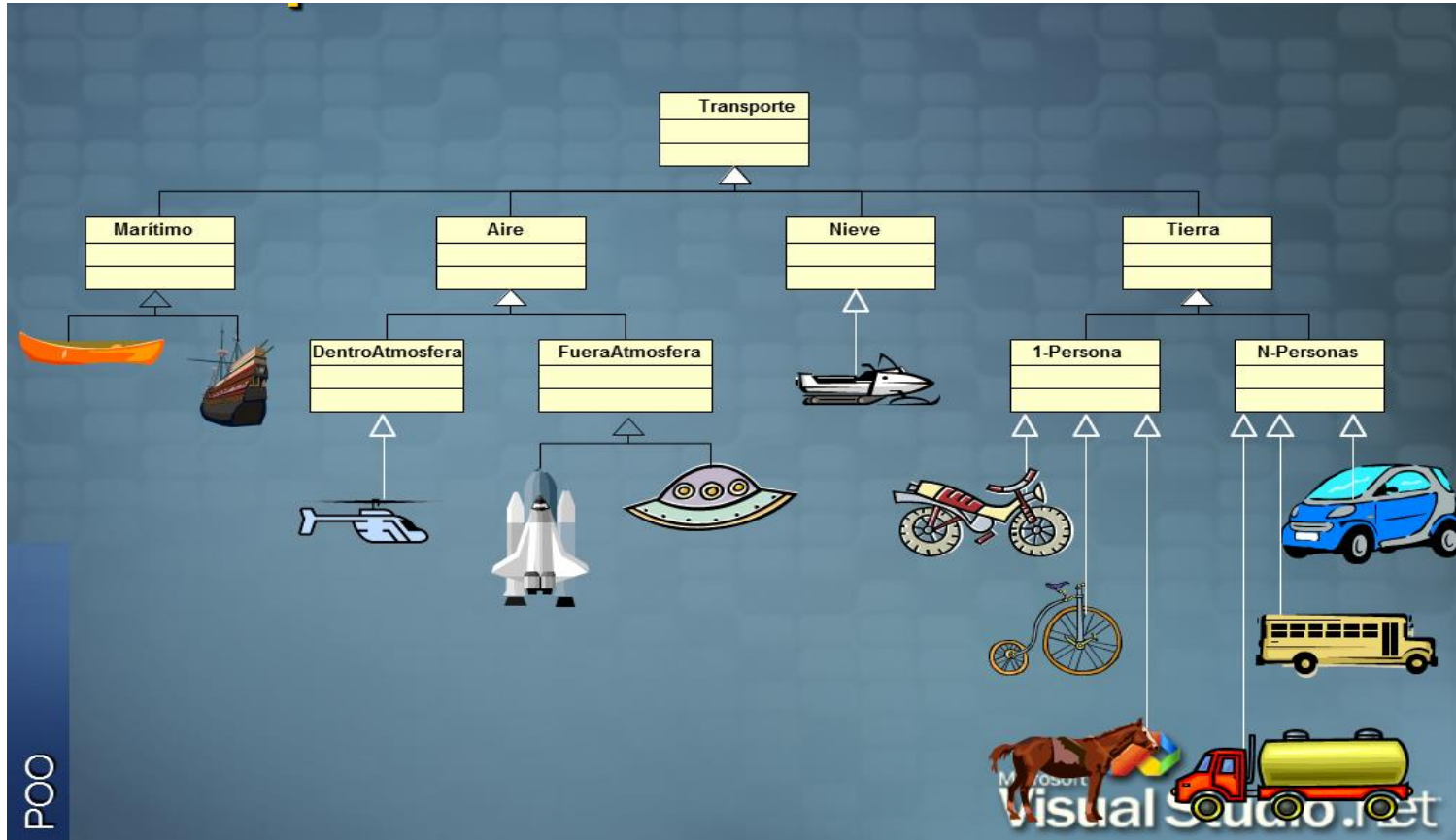
- Si se ha redefinido una propiedad o método de la clase padre en la clase hija, siempre es posible acceder a las propiedades o métodos de la clase padre, haciendo uso de la palabra reservada `parent`.
- Acceder a una propiedad de la clase padre que ya fue redefinida en la clase hija:

`parent::property;`

- Acceder a un método de la clase padre que ya fue redefinido en la clase hija:

`parent::method();`

# Herencia de clases (derivación)



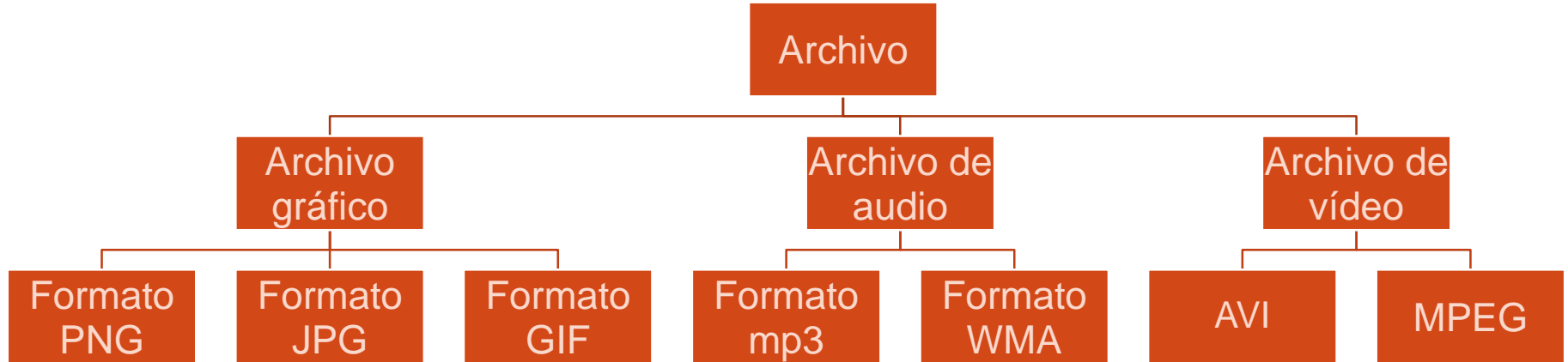
# Ejemplo de herencia (I)

- Intentaremos explicar la herencia con un caso particular. Imaginemos un objeto archivo. Para nosotros un archivo es un conjunto de bytes almacenados de alguna forma en el disco fijo de una computadora, que posee un cierto número de características, entre ellas: el nombre, la extensión, el tamaño en bytes, el propietario, los permisos de acceso, la fecha de creación y la de modificación.
- Las acciones que se pueden realizar con archivos son, entre otras: copiarlo, renombrarlo, borrarlo, etc.

# Ejemplo de herencia (II)

- Los tipos de archivo pueden ser muy diversos. Se puede tratar de un archivo de texto en caracteres ASCII, una imagen en formato GIF, JPG o PNG, un archivo de audio como mp3, wma, una película en formato DivX, MPEG, AVI, etc.
- Podemos imaginarnos este ejemplo en un esquema o diagrama de jerarquía, como se ilustra:

# Ejemplo de herencia (III)



# Clases y métodos finales (I)

- Las clases y métodos finales proporcionan al desarrollador un mecanismo de control sobre la herencia.
- Las propiedades, métodos y clases que se declaran como finales evitan que se pueda sobrecargar la clase, que se implemente un método de la clase padre en cualquier clase derivada o que se vuelva a definir una propiedad de la clase padre, también en sus clases derivadas.
- Para este propósito, sólo debe anteponerse la palabra reservada *final* al nombre de la clase, a la palabra reservada *function* o al nombre de la propiedad.

# Clases y métodos finales (II)

- Una clase declarada como **final** no puede tener subclases:

```
final class archivoPDF{  
    public function crearPDF(){//Lógica de la función}  
}
```



- Un método que se declara como **final** dentro de una clase, puede ser utilizado por las subclases o clases heredadas, pero no puede ser sobrescrito por estas. Esto significa que no puede volver a implementarse en las subclases.

```
class archivoMP3{  
    final public function playMP3(){//Lógica de la función}  
}
```



# Clases y métodos finales (III)

- Intentar derivar una clase que ya fue declarada con final producirá un error fatal que interrumpirá la secuencia de comandos PHP.
- Del mismo modo, si dentro de una clase derivada se intenta implementar un método que ya fue declarado como final dentro de su clase padre, también producirá un error fatal.

```
public final class A{
```

```
public class B extends A{
```

Final classes cannot be extended

❑ Class

```
public final class Math
```

❑ Method

```
public final double sqrt(int i);
```

❑ Attribute

```
public final float PI = 3.14;
```

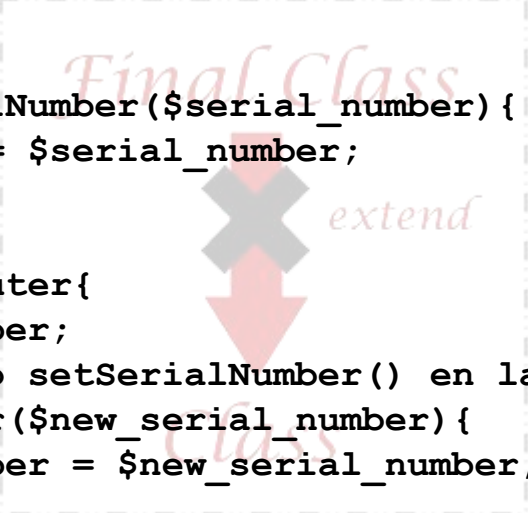
# Clases y métodos finales (IV)

- Ejemplo

```
class computer{
    private $serial_number;
    final function setSerialNumber($serial_number){
        $this->serial_number = $serial_number;
    }
}

class laptop extends computer{
    private $new_serial_number;
    //Redefiniendo el método setSerialNumber() en la clase derivada
    function setSerialNumber($new_serial_number){
        $this->new_serial_number = $new_serial_number;
    }
}

//Probando
$portable = new laptop();
//Cuando se intente ejecutar esta instrucción se arrojará un error fatal
$portable->setSerialNumber("1234aBCd");
```



# Clases abstractas (I)

- Otro de los recursos que proporciona mayor control sobre el proceso de herencia son las clases y métodos abstractos.
- Es común que encontremos situaciones donde una clase represente un concepto abstracto y, como tal, no debería ser posible concretarlo.
- De modo similar, en términos de programación orientada a objetos, tendría que ser posible modelar un concepto abstracto sobre el que no sea posible la creación de instancias del mismo.

# Clases abstractas (II)

- Esto significa que estas clases representarán conceptos abstractos que no pueden concretarse mediante instancias.
- Lo anterior significa, que como desarrolladores deberíamos poder modelar objetos abstractos impidiendo que se puedan crear instancias del mismo.
- Podemos entender mejor la abstracción, pensando en un concepto de la vida real como la comida. Todos sabemos lo que es la comida, pero ¿la hemos visto alguna vez?. Lo que si hemos visto es algún plato de pollo, carne, arroz, frijoles, etc.

# Clases abstractas (III)

- Sin embargo, el concepto de comida es, en si mismo, un concepto abstracto. Existe sólo como una generalización de cosas más específicas. Lo mismo sucede con conceptos como bebidas, postres, medicinas, etc.

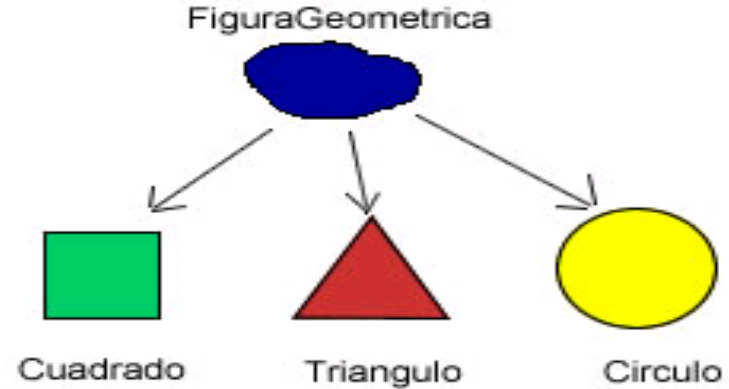
## Clases abstractas

```
abstract class Forma {  
    private $x, $y;  
  
    function __construct($x, $y) {  
        $this->x = $x; $this->y = $y;  
    }  
  
    abstract function area();  
  
    function mover($dx,$dy) {  
        $this->x+=$dx;  
        $this->y+=$dy;  
    }  
}
```

```
class Circulo extends Forma {  
    private $radio ;  
  
    function __construct($radio,$x,$y) {  
        parent::__construct($x,$y);  
        $this->radio = $radio;  
    }  
  
    function area() {  
        return pi() * pow($this->radio,2);  
    }  
}
```

# Clases abstractas (IV)

- La característica principal de una clase abstracta es que no puede ser instanciada.
- Una clase abstracta es utilizada en Programación Orientada a Objetos para crear objetos abstractos.
- Una clase abstracta es diseñada para proporcionar una super clase que representa un objeto abstracto que no puede ser concretado.



# Clases abstractas (V)

- Para crear una clase abstracta en PHP debe anteponerse la palabra reservada ***abstract*** delante de la palabra, también reservada, ***class***.
- Una clase abstracta, además de declarar e implementar propiedades y métodos, puede definir también métodos abstractos.
- Los métodos abstractos no se implementan (no contienen código alguno). Estos métodos deben ser implementados en las clases hijas o derivadas.
- También, es posible que en la clase hija, el método siga declarándose abstracto. Si esto ocurre la implementación de la clase podría realizarse en el siguiente nivel jerárquico.

# Clases abstractas (VI)

```
<?php
```

```
    abstract class number {
        private $value;
        abstract public function value();
        public function reset(){
            $this->value = NULL;
        }
    }

    //Clase que implementa el método abstracto
    class integer extends number {
        private $value;
        public function value(){
            return (int)$this->value;
        }
    }

    $entero = new integer(); //se creará el objeto entero
    $numero = new number(); //esto producirá un error
```

```
?>
```



# Polimorfismo

- Los lenguajes de programación orientados a objetos admiten el polimorfismo que consiste en que las clases pueden tener diferente comportamiento.
- Esto se realiza implementando un método que es capaz de ejecutarse de forma diferente dependiendo del tipo y la cantidad de argumentos que reciba.
- En una aplicación compleja este método podría estar implementado por objetos de distinta clase y, sin embargo, no sería relevante para la aplicación dónde está realmente implementado dicho método. Lo importante es obtener el resultado esperado.

# Interfaces

- Las interfaces son similares en algunos aspectos a las clases abstractas en el sentido que no pueden instanciarse.
- Las interfaces han sido diseñadas para asegurar la funcionalidad dentro de una clase, definiendo un conjunto de métodos que deben tener una clase para implantar dicha interfaz.
- Una interfaz, a diferencia de la clase abstracta, no puede ofrecer funcionalidad, ya que sólo puede declarar métodos, no implementarlos.
- Para declarar una interfaz debe utilizarse la palabra reservada *interface*.
- La clase que implementará un método de la interfaz debe declararse con la palabra reservada `implements nombre_interfaz`, justo a continuación del nombre de la clase.

# Ejemplo de interfaz

```
interface printable {  
    public function printme();  
}  
  
//Implantación de la interfaz  
class Integer implements printable {  
    private $value;  
    public function getValue(){  
        return (int)$this->value;  
    }  
    public function printme(){  
        echo (int)$this->value;  
    }  
}
```

# Manejo de excepciones (I)

- Las excepciones son un concepto relativamente nuevo en PHP, que permite activar un error, no fatal, en secuencias de comando PHP.
- El soporte para el manejo de excepciones en PHP, nos permite producir errores y controlarlos de formas más ventajosas que los métodos que existían en la versión PHP 4.
- Con PHP 4 había que utilizar funciones como `trigger_error()` y como `set_error_handler()` para activar errores y procesarlos, respectivamente.

# Manejo de excepciones (II)

- En la práctica, las excepciones son instancias de clase que contienen información sobre un error que se ha producido durante la ejecución de una secuencia de comando (*script*).
- PHP 5 proporciona internamente una clase denominada **Exception** que posee varios métodos que obtienen información de depuración mediante la cual se puede establecer las causas de un error.



# Clase Exception

```
class Exception {  
    //Propiedades  
    protected $message;  
    private $string;  
    protected $code;  
    protected $file;  
    protected $line;  
    private $trace;  
    //Métodos  
    function __construct($message = "", $code = 0);  
    function __toString();  
    public function getFile();  
    public function getLine();  
    public function getMessage();  
    public function getCode();  
    public function getTrace();  
    public function getTraceAsString();  
}
```

# Funcionamiento del manejo de errores

- Las excepciones en PHP contienen dos valores principales, que son: la cadena que describe el error que se ha producido y un código entero único asociado con ese error.
- Cuando se produce un error PHP asigna de forma automática a la excepción la línea dentro del ***script*** y el nombre del archivo donde se ha producido dicho error.
- Adicionalmente, se asigna una localización de pila que representa la ruta de acceso a la ejecución que ha resultado en error.

# Lanzar una excepción

- Las excepciones son objetos de la clase predefinida ***exception*** (o de una clase derivada de esta) que se lanzan mediante la palabra reservada `throw`, utilizando una instrucción como la siguiente:  

```
throw new Exception($message, $coderror);
```
- La sentencia **`throw`** termina de forma súbita la ejecución del método donde se encuentre y generan un objeto de la clase `Exception` disponible en el contexto del objeto.



# Ejemplo de una excepción lanzada

```
<?php
```

```
class throwExample {
```

```
    public function makeError() {
```

```
        throw new Exception("Esto ha generado una excepcn.");
```

```
    }
```

```
}
```

```
$inst = new throwExample();
```

```
$inst->makeError();
```

```
?>
```

# Estructura try/catch

- La gestión de excepciones en PHP se realiza mediante la utilización de una estructura, nueva en PHP 5, conocida como try/catch.
- Su funcionamiento es relativamente simple, dentro del bloque try se colocan las instrucciones que pueden llegar a producir una excepción.
- Cada bloque **try**, debe contener, al menos, un bloque **catch**, con los cuales se gestionarán las excepciones.

```
try {  
    // do something  
} catch (error)  
    // about errors  
}
```

# Sintaxis de la estructura *try/catch*

- Un bloque *try/catch* tiene la siguiente sintaxis:

```
try {  
    //código que se intentará ejecutar y que  
    //puede generar una excepción  
}  
catch(classException1 $e1){  
    //Procesamiento de las excepciones de  
    //classException1  
}  
[catch(classException2 $e2){  
    //Procesamiento de las excepciones de  
    //classException2  
}]
```

# Ejemplo con la estructura *try/catch*

```
<?php
class myException extends Exception{
    /* Hereda todas las propiedades y métodos de la clase Exception */
}
class anotherException extends Exception{
    /* De nuevo, hereda todas las propiedades y métodos de la clase
Exception */
}
Class throwExample{
    public function makeMyException(){
        throw new myException();
    }
    public function makeAnotherException(){
        throw new anotherException();
    }
    public function makeError(){
        throw new Exception();
    }
} ...
```

# Ejemplo con la estructura *try/catch*

```
...
$inst = new throwExample();
try{
    $inst->makeMyException();
} catch(myException $e){
    echo "'myException' capturada\n";
}
try{
    $inst->makeAnotherException();
} catch(myException $e){
    echo "'myException' capturada\n";
} catch(anotherException $e){
    echo "'anotherException' capturada\n";
}
try{
    $inst->makeError();
} catch(myException $e){
    echo "'myException' capturada\n";
} catch(anotherException $e){
    echo "'anotherException' capturada\n";
}
}
```

?>

FIN

