UCH UNIVERSIDAD DE CIENCIAS Y HUMANIDADES

PROGRAMACIÓN II

SEMANA 09
CONSTRUCTORES Y DESTRUCTORES

Eric Gustavo Coronel Castillo
ecoronel@uch.edu.pe
gcoronelc.blogspot.com

CONSTRUCTORES Y DESTRUCTORES

Toda clase está conformada por atributos y métodos; sin embargo, existen métodos especiales que permiten inicializar un objeto, y otro método especial que permite liberar los recursos que está utilizando un objeto.

Índice

1 Constructores

1.1 Definición e Implementación

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado. Esta inicialización se lleva a cabo invocando en un método especial denominado constructor. Esta invocación se realiza cuando se utiliza el operador new.

Los constructores tienen algunas características especiales:

- El nombre del constructor es __construct.
- Puede recibir argumentos como cualquier otro método.

El constructor no es un miembro más de una clase. Sólo es invocado cuando se crea el objeto con el operador new. No puede invocarse explícitamente en ningún otro momento.

Sintaxis

```
[TipoAcceso ] function __construct ( [ ListaDeParámetros ] ) {
   // Implementación
}
```

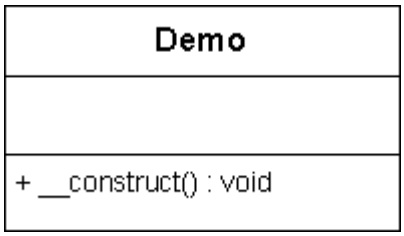


Figura 1 Representación UML del constructor de una clase.

En la Figura 1 tenemos la representación UML del constructor de una clase, la palabra void significa que no retorna ningún valor.

A continuación tenemos su implementación:

```
class Demo {

    // Constructores

    public __construct ( ) {

        // Implementación

    }

    // Implementación de la clase

} // Clase Demo
```

Al crear un objeto con el operador new se ejecuta el constructor.

```
$objeto = new Demo();
```

Los constructores padres no son llamados implícitamente si la clase hija define un constructor. Para poder ejecutar el constructor de la clase padre, se debe hacer una llamada a **parent::__construct()** dentro del constructor de la clase hija.

Ejemplo 1

Se necesita una clase para realizar operaciones con una base de datos MySQL; es requisito que en el instante de crear el objeto se conecte con la base de datos automáticamente.

Se utilizará la base de datos de recursos humanos **(rh)** para ilustrar el ejemplo y se construirá un método que permita ejecutar una consulta.

La solución para este requerimiento es hacer una clase que en su constructor realice la conexión con la base de datos; la representación UML se muestra en la Figura 2.

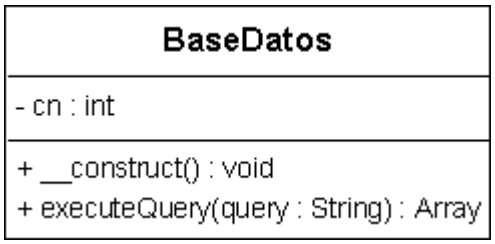


Figura 2 Representación UML de la clase **BaseDatos**.

A continuación tenemos su implementación:

Archivo: BaseDatos.php

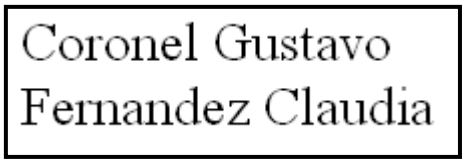
```
<?php

class BaseDatos {

   private $cn = null;

   public function __construct() {
      $this->cn = mysql_connect("localhost", "root", "admin");
      mysql_select_db("rh",$this->cn);
   }

   public function executeQuery( $query ) {
      $rs = mysql_query( $query, $this->cn );
      $lista = array();
      while( $rec = mysql_fetch_assoc($rs) ){
         $lista[] = $rec;
      }
      return $lista;
   }

}

?>
```

Para probar la clase **BaseDatos** utilizaremos el siguiente programa:

Archivo: Prueba.php

```
<?php

   require_once 'BaseDatos.php';

   $bd = new BaseDatos();
   $query = "select * from empleado where iddepartamento = '100'";
   $lista = $bd->executeQuery($query);

   foreach ($lista as $rec) {
      echo("{$rec["apellido"]} {$rec["nombre"]}<br>");
   }

?>
```

Al ejecutar éste programa obtenemos el siguiente resultado:



1.2 Sobrecarga del Constructor

PHP no implementa la sobrecarga, pero podemos simularla.

Supongamos que en la clase **BaseDatos** del Ejemplo 1 queremos tener dos opciones, que se describen a continuación:

1. En la primera opción queremos ejecutar el constructor sin parámetros, en cuyo caso asumirá datos por defecto.

2. En la segunda opción queremos pasarle los parámetros de conexión.

Para tener estas dos posibilidades definimos el constructor con los parámetros de conexión, pero con valores por defecto, y cuando creamos el objeto utilizando el operador **new** podemos pasar o no parámetros,

El nuevo constructor quedaría así:

```
public function __construct($server = null, $user = null, $pwd = null) {
   if( is_null($server) || is_null($user) || is_null($pwd) ) {
      $this->cn = mysql_connect("localhost", "root", "admin");
   } else {
      $this->cn = mysql_connect($server, $user, $pwd);
   }
   mysql_select_db("rh",$this->cn);
}
```

Podemos instanciar la clase **BaseDatos** de la siguiente manera:

```
$bd = new BaseDatos();
```

O de esta otra:

```
$bd = new BaseDatos("localhost", "root", "admin");
```

2 Destructores

Un destructor es un método especial que es ejecutado cada vez que se destruye (se elimina de la RAM) un objeto.

Es de visibilidad pública (public), no tiene parámetros, ni valor de retorno y se utiliza para liberar recursos utilizados por el propio objeto.

Su sintaxis es:

```
[public] function __destruct ( ) {

   // Implementación

}
```

El método destructor será llamado tan pronto como todas las referencias a un objeto en particular sean removidas, cuando el objeto sea explícitamente destruido, o cuando finalice la ejecución del programa.

Como sucede con el método constructor, el método destructor de la clase padre no será llamado implícitamente. Para ejecutar un destructor padre, se debe hacer una llamada explícita a **parent::__destruct()** en el cuerpo del destructor.

> **Nota**
>
> Para destruir un objeto explícitamente debe utilizar la función **unset()**.

> **Nota**
>
> Intentar disparar una excepción desde un destructor (llamado durante la finalización del script) produce un error fatal.

Esperar a que el mismo sistema ejecute el método destructor para que se liberen los recursos utilizados por el objeto puede resultar muy costoso. Por ejemplo, en la clase **BaseDatos** del Ejemplo 1 podrías tener la conexión con la base de datos activa por mucho tiempo causando problemas de acceso a otros usuarios.

Es posible implementar un método público para que sea invocado cada vez que se necesite liberar recursos utilizados por el objeto; muchas veces este método es llamado **dispose**.

A continuación tenemos la representación UML de los métodos constructor, destructor y dispose para una clase:

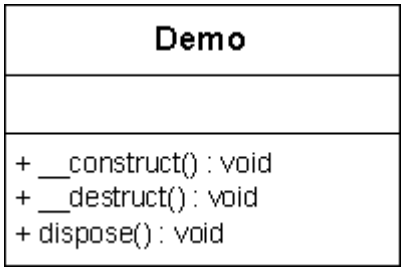


Figura 3 Representación UML de los métodos **__construct, __destruct** y **dispose**.

A continuación tenemos su implementación:

```
class Demo {

    // Constructor
    public function __construct() {

        // Implementación

    }

    // Destructor
    public function __destruct() {

        // Implementación

    }

    // Método para liberar recursos
    public function dispose() {

        // Implementación

    }

} // Demo
```

Ejemplo 2

Se necesita implementar los métodos __**destruct** y **dispose** de la clase **BaseDatos** del Ejemplo 1.

La representación UML se muestra en la Figura 4.

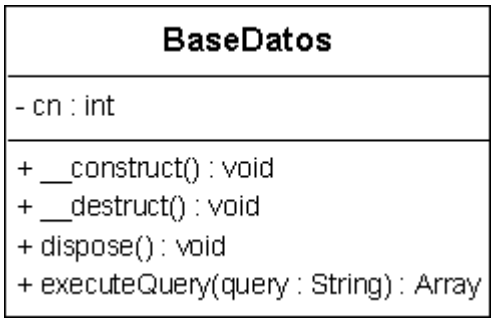


Figura 4 Representación UML de la clase BaseDatos.

A continuación tenemos su codificación:

```
<?php

class BaseDatos {

   private $cn = null;
   private $liberar = TRUE;
   const LOGFILE = "prueba.log";

   public function __construct() {
      $this->cn = mysql_connect("localhost", "root", "admin");
      mysql_select_db("rh",$this->cn);
      error_log("Conexión Ok.\n", 3, self::LOGFILE);
   }

   public function  __destruct() {
      if( $this->liberar ) {
         $this->dispose();
      } else {
         error_log("Conexión ya fué liberada.\n",3,self::LOGFILE);
      }
   }

   public function dispose() {
      if( $this->liberar ) {
         if( $this->cn ) {
            mysql_close( $this->cn );
            $this->cn = null;
            error_log("Conexión Liberada.\n", 3, self::LOGFILE);
         }
```

```
            $this->liberar = FALSE;
        }
    }

    public function executeQuery( $query ) {
        $rs = mysql_query( $query, $this->cn );
        $lista = array();
        while( $rec = mysql_fetch_assoc($rs) ){
            $lista[] = $rec;
        }
        return $lista;
    }

}

?>
```

3 Alcance de Instancia y Alcance de Clase

3.1 Introducción

3.1.1 Alcance de Instancia

Son campos y métodos que necesitan ser invocados por una instancia, es decir, por un objeto, ya que se crean en el objeto.

```
$nombreObjeto->nombreCampo

$nombreObjeto->nombreMétodo( ... )
```

3.1.2 Alcance de Clase

Son campos y métodos que no requieren crear una instancia (objeto) para ser invocados, basta con anteponer el nombre de la clase, ya que se encuentran en la clase y no en el objeto.

```
NombreClase::nombreCampo

NombreClase::nombreMétodo( ... )
```

En la programación orientada a objetos se les conoce como campos y métodos estáticos (static), en el caso de ser un campo estático mantiene su valor para cada instancia de la clase; de ser público el campo podemos acceder incluso desde cualquier otra clase.

3.2 Implementación

Su sintaxis es para los campos es:

```
[private|public|protected] static $nombreCampo [= valor];
```

Su sintaxis para los métodos es:

```
[private|public|protected] static function nombreMétodo( [parámetros] ) {

   // Implementación

}
```

La presencia de la palabra clave static determina que la declaración tiene alcance de clase.

Cuando crea una instancia de la clase, no se crean copias de las variables estáticas; en vez de ello, todas las instancias comparten las mismas variables estáticas, es decir el valor es compartido.

Los métodos de clase solo pueden acceder a campos y métodos de clase y no a campos y métodos de instancia; y no se puede usar la palabra clave $this en un método estático.

A continuación tenemos la representación UML de una clase con campos y métodos de instancia y clase.

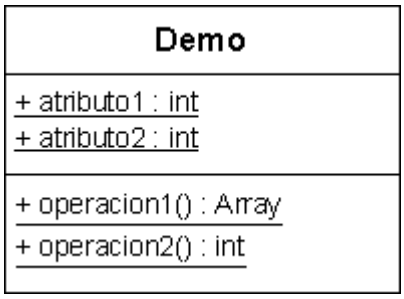


Figura 5 Alcance de Instancia y de Clase.

A continuación tenemos su implementación:

```
class Demo {

   public static $atributo1 = 0;
   public static $atributo2 = 0;

   public static function  operacion1() {

      // Implementación

   }

   public static function  operacion2() {

      // Implementación

   }

}
```

3.3 Acceso a los Miembros Declarados en una Clase

3.3.1 Alcance de Instancia

Para acceder a los miembros de instancia desde la misma clase se debe utilizar la variable **$this** siguiendo la siguiente sintaxis:

```
$this->nombreCampo

$this->nombreMétodo( ... )
```

Fuera de la clase debemos utilizar la variable que hace referencia al objeto, por ejemplo si instanciamos la clase **Demo**:

```
$obj = new Demo();
```

Para acceder a sus miembros de instancia debe utilizar la siguiente sintaxis:

```
$obj->nombreCampo

$obj->nombreMétodo( ... )
```

3.3.2 Alcance de Clase

Para acceder a los miembros de clase desde la misma clase se debe utilizar la palabra clave **self** siguiendo la siguiente sintaxis:

```
self::$nombreCampo

self::nombreMétodo( ... )
```

Desde fuera de la clase debemos utilizar el nombre de la clase siguiendo la siguiente sintaxis:

```
NombreClase::$nombreCampo

NombreClase::nombreMétodo( .. )
```

4 Constantes de Clase

Es posible definir valores constantes en una clase. Las constantes se diferencian de las variables en que no se usa el símbolo dólar ($) para declararlas o usarlas.

Sintaxis:

```
const NOMBRE = valor;
```

Ejemplo:

```
const PROVEEDOR = "PeruDev";
```

Su compartimiento es similar a los campos de clase (estáticos) con visibilidad pública, quiere decir que para acceder a las constantes desde la misma clase debemos utilizar la palabra clave self y fuera de la clase debemos utilizar el nombre de la clase.

Como los miembros estáticos, a los valores constantes no se puede acceder desde una instancia de la clase donde se define.

El valor debe ser una **expresión constante**, no puede ser una variable, un miembro de una clase, resultado de una operación matemática ni la llamada a una función.

5 Proyectos Resueltos

5.1 Cuatro Operaciones

5.1.1 Requerimiento de Software

El área de matemáticas del colegio **"Los Amautas"** necesita de una librería con métodos de clase para las cuatro operaciones básicas: suma, resta, multiplicación y división.

Esta librería deberá ser incorporada en sus proyectos de construcción de software educativo en el área de matemáticas.

5.1.2 Abstracción

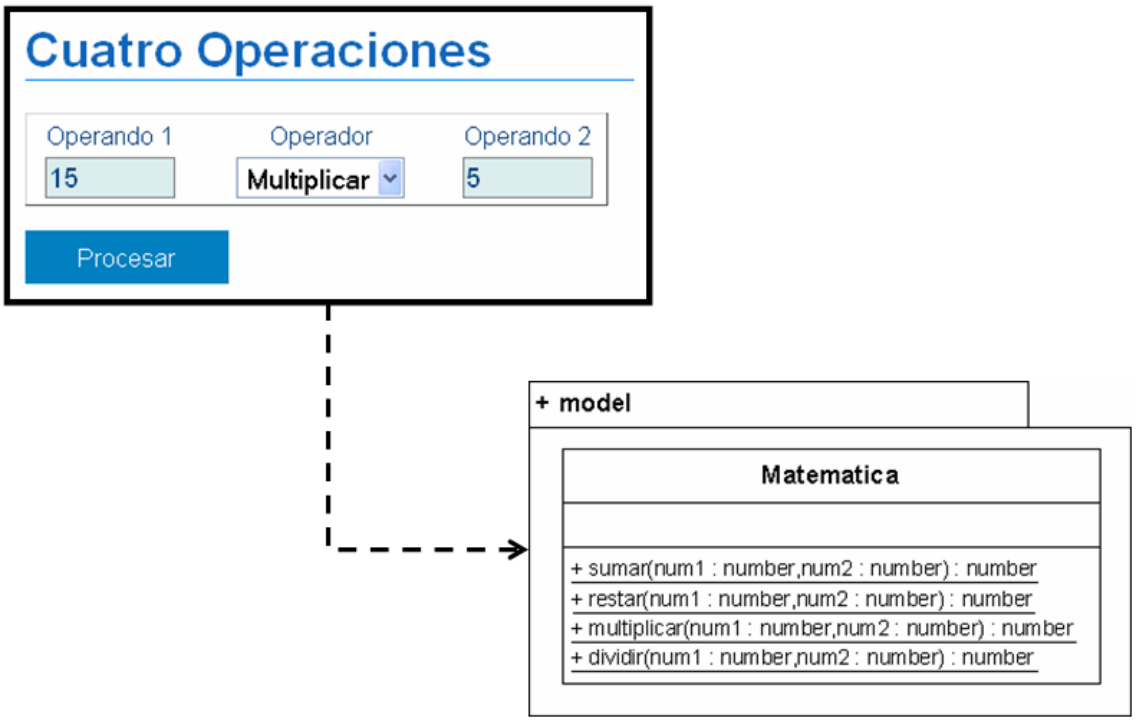
La solución planteada considera una clase de nombre **Matematica** con cuatro métodos de clase, un método para cada operación.

Método	Descripción
sumar()	Retorna la suma de dos números.
restar()	Retorna la resta de dos números.
multiplicar()	Retorno el producto de dos números.
dividir()	Retorna la división de dos números.

5.1.3 Diagrama de Clases

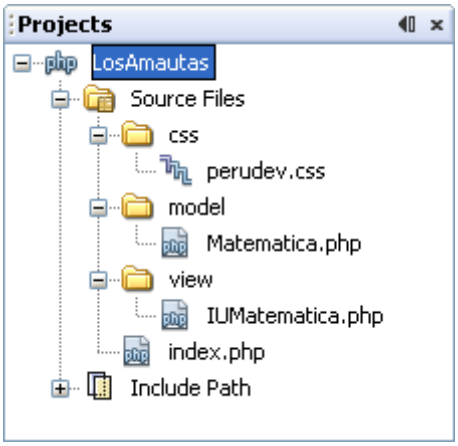
En este caso tenemos un programa PHP de nombre **IUMatematica.php** que nos servirá para ilustrar la funcionalidad de la clase Matemática, este programa se encuentra en la carpeta **view**.

El programa **IUMatematica.php** hace uso de la clase **Matematica**, y gráficamente lo representamos de la siguiente manera:



5.1.4 Estructura del Proyecto en NetBeans

Como se puede apreciar en el gráfico de la derecha, tenemos tres carpetas, en la carpeta **css** tenemos la hoja de estilos, en la carpeta **model** ubicamos la clase **Matematica**, y en la carpeta **view** el programa **IUMatematica.php**.



En la raíz del proyecto tenemos el programa **index.php**, éste es el que inicia la ejecución del proyecto, y en este caso es hacer un redireccionamiento al programa **IUMatematica.php**.

5.1.5 Codificación de la clase Matematica

Archivo: Matematica.php

```
<?php

class Matematica {

   public static function  sumar( $num1, $num2) {
      $rpta = $num1 + $num2;
      return $rpta;
   }

   public static function  restar( $num1, $num2) {
      $rpta = $num1 - $num2;
      return $rpta;
   }

   public static function  multiplicar( $num1, $num2) {
      $rpta = $num1 * $num2;
      return $rpta;
   }

   public static function  dividir( $num1, $num2) {
      $rpta = $num1 / $num2;
      return $rpta;
   }

}

?>
```

5.1.6 Codificación de la Interfaz de Usuario IUMatematica

Archivo: C:\PHP100\cap11\LosAmautas\view\IUMatematica.php

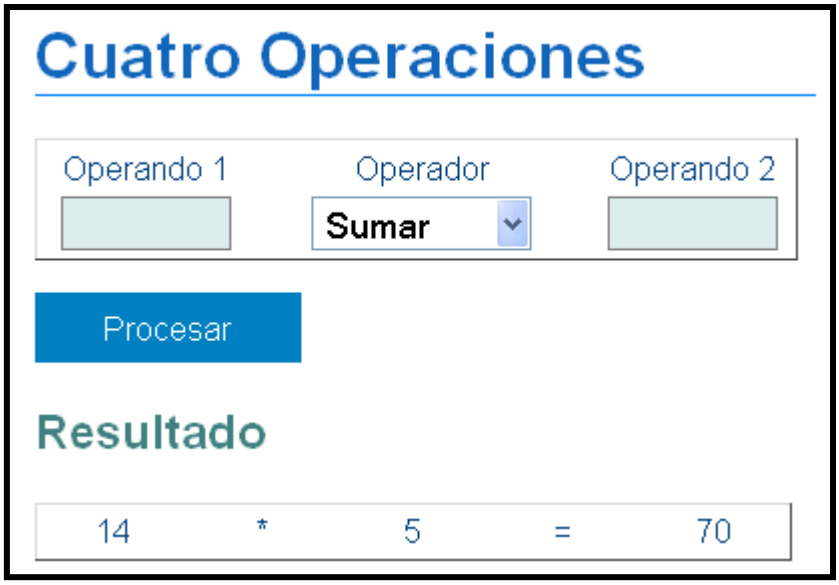
```
<?php
require_once '../model/Matematica.php';
$resultado = FALSE;
if( isset( $_REQUEST["procesar"] ) ) {
   // Datos
   $num1 = $_REQUEST["num1"];
   $num2 = $_REQUEST["num2"];
   $ope = $_REQUEST["oper"];
   // Proceso
   switch ($ope) {
      case "+":
         $rpta = Matematica::sumar($num1, $num2);
         break;
      case "-":
         $rpta = Matematica::restar($num1, $num2);
         break;
      case "*":
         $rpta = Matematica::multiplicar($num1, $num2);
         break;
      case "/":
         $rpta = Matematica::dividir($num1, $num2);
         break;
   }
   $resultado = TRUE;
}

?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
      <link rel="stylesheet" type="text/css" href="../css/perudev.css"/>
      <title>Cuatro Operaciones</title>
   </head>
   <body>
      <h1>Cuatro Operaciones</h1>
      <form name="form1" method="post" action="IUMatematica.php">
         <table width="286" border="1">
            <tr>
               <td width="74" align="center">Operando 1</td>
               <td width="120" align="center">Operador</td>
               <td width="70" align="center">Operando 2</td>
            </tr>
            <tr>
               <td align="center">
```

```
                <input name="num1" type="text" class="fieldEdit"
                  id="num1" size="5" maxlength="5">
              </td>
              <td align="center">
                <select name="oper" id="oper">
                  <option value="+">Sumar</option>
                  <option value="-">Restar</option>
                  <option value="*">Multiplicar</option>
                  <option value="/">Dividir</option>
                </select>
              </td>
              <td align="center">
                <input name="num2" type="text" class="fieldEdit"
                  id="num2" size="5" maxlength="5">
              </td>
            </tr>
          </table>
          <p>
            <input name="procesar" type="submit" class="button"
              id="procesar" value="Procesar">
          </p>
        </form>
        <?php if( $resultado ) { ?>
        <div id="resultado">
          <h2>Resultado</h2>
          <table width="284" border="1">
            <tr>
              <td width="50" align="center">
                <?php echo($num1) ?>
              </td>
              <td width="50" align="center">
                <?php echo($ope) ?>
              </td>
              <td width="50" align="center">
                <?php echo($num2) ?>
              </td>
              <td width="50" align="center">=</td>
              <td width="50" align="center">
                <?php echo($rpta) ?>
              </td>
            </tr>
          </table>
        </div>
        <?php } ?>
    </body>
</html>
```

5.1.7 Ejecución del Proyecto

Cuando ejecutamos el proyecto inicialmente tenemos la siguiente interfaz:

Después de ingresar los datos y hacer clic en el botón **Procesar** obtenemos el siguiente resultado:



Luego usted puede ingresar otros valores para hacer otra operación.

6 Proyectos Propuestos

6.1 Librería de Matemáticas

El colegio **"Ángeles del Cielo"** está solicitando un programa en PHP para que los alumnos de primaria verifiquen sus ejercicios de matemáticas referidos a:

- Calculo de factorial
- Calculo del MCD y MCM
- La serie de Fibonacci
- Número primo

La programación de estos cálculos matemáticos debe estar implementados como métodos de clase en una clase de nombre **Matematica**.

Se pide plantear la solución e implementarla aplicando los conceptos de POO.

6.2 Tipo de Cambio

La casa de cambio **"Dinero Express"** necesita de una aplicación en PHP que permita a sus empleados agilizar el cálculo del cambio de dinero de:

1. Soles a Dólares y viceversa.
2. Soles a Euros y viceversa.

Se pide plantear la solución e implementarla aplicando los conceptos de POO.