Parte VI

Polimorfismo

Polimorfismo

DEFINICIÓN: El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase (atributo, parámetro, declaración local o elemento de un vector) acepta direcciones de objetos de dicha clase y de sus clases derivadas (hijas, nietas,…).

Por ejemplo, la clase Jaula acepta Animales, pero nosotros lo que metemos en la jaula es un Perro, un Gato,…

El polimorfismo se expresa aceptando objetos de un elemento superior de la jerarquía de herencia.

Polimorfismo en PHP

En PHP, el polimorfismo no tiene mucho sentido, ya que de por sí es un lenguaje débilmente tipado. Por ello podríamos decir que el polimorfismo está en todas partes en el lenguaje PHP. Es una técnica más útil en los lenguajes fuertemente tipados como Java.

```
$polimorfico = array("dato" => "hola",
 "otro" => 22,
 "si"   => true,
 "muchas caras" => new Casa(),
 "cualquier cosa" => new Torpedo()
);
```

Polimorfismo

Pero aunque PHP es débilmente tipado, precisamente en aquellos métodos que reciben como parámetros objetos PHP permite especificarlo, para que así no podamos admitir como valor del parámetro cualquier cosa.

Por ejemplo, en la clase Jaula tenemos una propiedad llamada $contenido, que lo que pretendemos que sea es un objeto de la clase mamífero. Y disponemos del método meter() que nos permite introducir en una Jaula un Mamífero. Así pues:

```
public function meter(Mamifero $animal){
  $this->contenido = $animal;
}
```

Si no escribiéramos Mamífero delante de $animal (que es una referencia al tipo del objeto), PHP permitiría hacer la siguiente llamada:

```
$jaula->meter("Hola, soy un string");
```

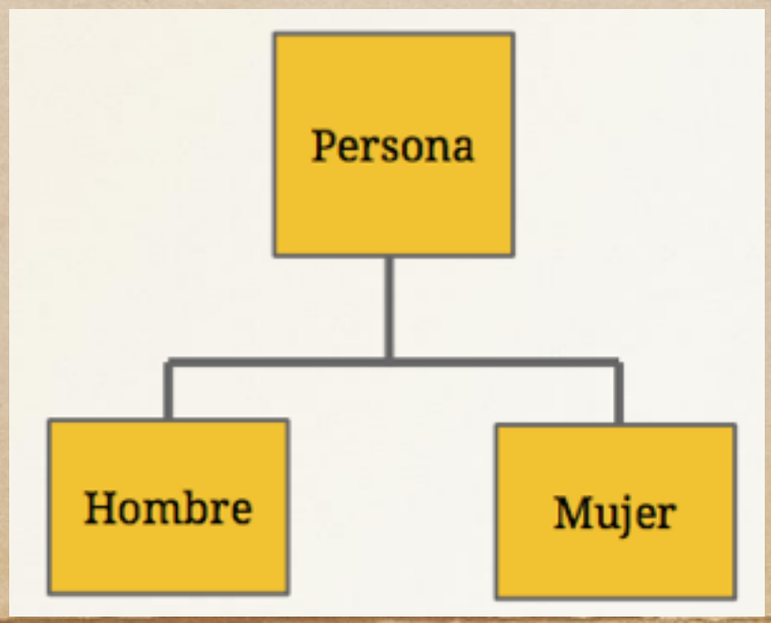
Mientras que al haberlo puesto nos da un error fatal.

Herencia y Polimorfismo

Con la incorporación del polimorfismo, tiene sentido declarar referencias a clases abstractas con la intención de almacenar direcciones de objetos de clases concretas derivadas, NO de clases abstractas que no son instanciables.

En el polimorfismo trabajamos con conceptos como herencia y abstracción, que también tienen sentido en la programación PHP.

```
public function contratar(Persona $persona){…}
```



Ejemplo Clase Persona

```
abstract class Persona
{
    private $nombre;

    function __construct($nombre){
        $this->nombre = $nombre;
    }
    public function saludar(){
        echo "Hola, soy $this->nombre";
    }
    public abstract function mear();
}

class Hombre extends Persona
{
    public function mear(){
        echo "Me aproximo al urinario, apunto y me relajo para vaciar la vejiga";
    }
}

class Mujer extends Persona
{
    public function mear(){
        echo "Me siento en el retrete y me relajo para vaciar la vejiga";
    }
}
```

Continuación ejemplo

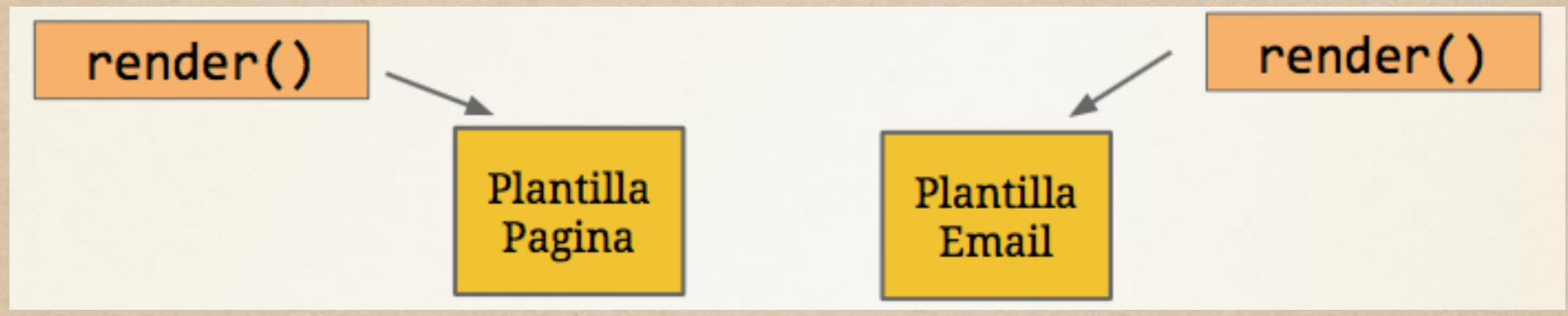
```
class Casa
{
    private $habitantes = [];

    public function habitar($persona){
        $this->habitantes[] = $persona;
        $persona->saludar();
    }
    public function pasear(){
        foreach ($this->habitantes as $cual) {
            $cual->mear();
        }
        //y otras cosas del paseo
    }
}
//A partir de aquí podría ser index.php con los include correspondientes.
$miguel = new Hombre("miguel");
$sara = new Mujer("sara");

$casa = new Casa();
$casa->habitar($sara);
$casa->habitar($miguel);
//$casa->habitar(new Perro); //Esto da un error
$casa->pasear();
class Perro
{
}
```

Polimorfismo en PHP

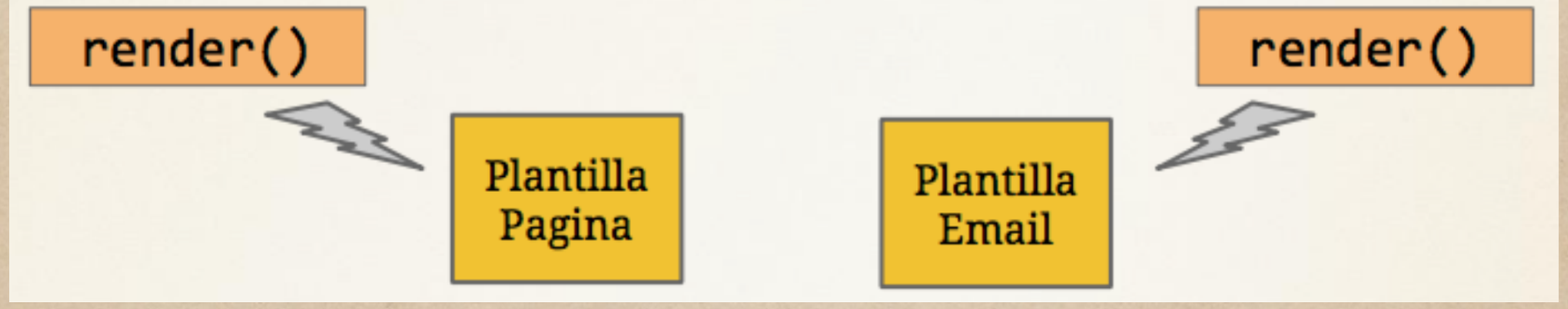
- Plantilla tiene un método abstracto render().
- Ese método se debe implementar en todos los tipos de plantillas que queramos realizar.



Polimorfismo en PHP

```
$saludo = new PlantillaHTML($html);
$registro = new PlantillaEmail($html,"aa@aa.aa","Gracias");

$saludo->render();
$registro->render();
```



Como estropearlo todo

Cuando trabajamos con un esquema de polimorfismo hay algo que NUNCA debemos hacer:

```
if(is_a($objPlantilla, "PlantillaHTML")){
  //Hacer algo sabiendo que es una Plantilla HTML
}elseif(is_a($objPlantilla, "PlantillaEmail")){
  //hacer algo sabiendo que es una Plantilla de Email
}
```

Como estropearlo todo

Cuando trabajamos con un esquema de polimorfismo hay algo que NUNCA debemos hacer:

```
if(is_a($objPlantilla, "PlantillaHTML")){
  //Hacer algo sabiendo que es una Plantilla HTML
}elseif(is_a($objPlantilla, "PlantillaEmail")){
  //hacer algo sabiendo que es una Plantilla de Email
}
```

Lo que no debemos hacer NUNCA es preguntar a un objeto de que tipo es. Para eso tenemos la clase Plantilla, que como clase padre abarca a sus hijas con métodos comunes.

Si en algún momento añadiésemos la clase PlantillaExcel, tendríamos que revisar todo el código como el anterior para añadir otro **elseif**, en cambio, tal y como vimos con render() si aprovechamos las clases abstractas y el polimorfismo, no sería necesario.

Ejercicios

Select con botones de radio.

Queremos hacer una nueva clase que hereda de Select. Esa clase se llamaría SelectRadio y lo que implementa es el comportamiento de un campo Select, donde se puede seleccionar una opción entre varias, pero por medio de botones de radio. En esta clase, al imprimir el Select, en lugar de usar la etiqueta SELECT de HTML usamos etiquetas INPUT type="radio".

Ejercicios

Línea de Formulario.

Hacer una clase llamada "LineaFormulario". Esa clase lo que hará será imprimir una línea de formulario como esta:

```
<p>
 <label>Nombre del Campo</label>
 <span>#campo#</span>
</p>
```

siendo #campo# un campo del formulario. La idea es usar los select que hemos creado. La clase deberá recibir en su constructor el "nombre del campo", para la etiqueta label y el objeto select que desee mostrar en #campo#. Podrá ser un objeto de la clase padre Select o cualquiera de las clases hijas que hemos ido proponiendo.

Ejercicios

Select "Universal".

Queremos crear una clase nueva llamada "SelectUniversal" que coordina el trabajo entre las distintas clases de Select que se implementaron. Esta clase, en su constructor recibe un array con opciones y, dependiendo de esas opciones, puede instanciar un objeto Select de una clase o de otra, según la que sea más conveniente para resolver la necesidad. Al imprimir el Select desde "SelectUniversal" se debe delegar el trabajo en el select instanciado. Hay que darse cuenta que este ejercicio no implica que esta nueva clase deba tener una relación de herencia con las otras ya creadas. Lo que queremos implementar es una relación de uso.

Para que la clase SelectUniversal sepa que tipo de select debe instanciar tiene que deducirlo del array de opciones.

Interfaces

Por la relajación de tipos que tiene PHP, podemos implementar un esquema típico de polimorfismo mediante otras herramientas para producir herencia, como son las **interfaces** (y los **traits** que los veremos más adelante).

Las **interfaces** de objetos permiten crear código con el cual especificamos qué métodos deben ser implementados por una clase, sin tener que definir cómo estos métodos son manipulados. Todos los métodos declarados en las **interfaces** deben ser públicos, ya que esta es su naturaleza.

Por tanto, podemos entender las **interfaces** como esquemas o guiones o un contrato a seguir por las clases que las implementen.

Las clases pueden implementar más de una interfaz, simplemente separándolas mediante comas en su declaración.

Síntaxis de las Interfaces

```
interface iInterruptor
{
 public function enciende();
 public function apaga();
}
```

Las interfaces permiten definir también constantes, pero no atributos. Y tampoco podemos definir código en sus métodos.

Como vemos en la definición, todos los métodos definidos deben ser públicos.

Interfaces

Las clases que implementan interfaces tienen la siguiente sintaxis:

```
class Bombilla implements iInterruptor
{
  //…
}
class TV implements iInterruptor
{
  //…
}
```

Si no definimos todos los métodos de la interfaz, estamos obligados a declarar la clase como abstracta.

También, una clase puede implementar varias interfaces:

```
class Conductor implements iConducir, iReparar {
}
```

Herencia de Interfaces

Las interfaces pueden extenderse, con **extends**, por un mecanismo de herencia.

```
interface iOrganizadorDiario{
 public function agendaDia();
 public function reservaHorario($hora);
}
```

```
interface iOrganizadorMensual extends iOrganizadorDiario{
 public function agendaSemana();
 public function agendaMes();
}
```

Aquella clase que implemente la interfaz iOrganizadorMensual tendrá que implementar sus dos métodos y también los otros dos heredados de la interfaz padre. Si alguno de ellos no lo implementara hemos de recordar que se tendrá necesariamente que declarar como abstracta.

Ejemplo: Carrito de la Compra

Vamos a crear un carrito de la compra que acepta objetos polimórficos:

- Productos.
- Descuentos (códigos promocionales).
- Packs.

Debemos tener en cuenta lo siguiente:

- El carrito se debe almacenar en una variable de sesión para recordar su valor a lo largo de toda la visita del usuario.
- Cada tipo de producto se muestra con ligeras diferencias, ya que un producto y un descuento, o un pack, no se pueden mostrar de la misma manera.
- Generaremos el polimorfismo a través de una interfaz, dado que los objetos que pretendemos meter en el carrito no pertenecen a un sistema de herencia claro. ¿Un descuento es un producto?

Carrito de la compra

Como hemos hecho hasta ahora en otros ejemplos desarrollados anteriormente, vamos a ir obteniendo diferentes versiones de lo que nos piden, cada una de ellas funcional pero mejorando la anterior. Así vamos viendo la evolución en el desarrollo del software.

Vamos a empezar la creación de nuestro carrito sin tener en cuenta el polimorfismo, ya que solo vamos a tener dentro productos tradicionales (de momento sin packs ni descuentos).

Empezaremos creando la clase Carrito que contendrá elementos (objetos) de la clase Productos. De momento guardaremos todas las clases en el archivo carrito.php.

Utilizaremos el archivo enlace.css para darle algo de formato a lo que vamos a ir haciendo.

La clase carrito tiene una propiedad **$productos** que contiene los productos que vamos añadiendo. Como puede ser un número indeterminado, lo definimos como array.

Tendremos el método meter(), para añadir productos al carrito, y mostrar(), para visualizar lo que contiene. Como primera aproximación es suficiente.

```
class Carrito
{
   private $productos = [];

   public function meter($producto){
   }

   public function mostrar(){
   }
}
```

```php
class Carrito
{
    private $productos = [];

    public function meter($producto){
        $this->productos[] = $producto;
    }

    public function mostrar(){
        $total = 0;
        $totalIva = 0;
        echo '<div class="carrito">';
        foreach ($this->productos as $prod) {
            echo '<article class="lineacarrito">';
            echo $prod->mostrar();
            echo '</article>';
            $total += $prod->precio();
            $totalIva += $prod->precioIva();
        }
        echo '<div class="totalcarrito">TOTAL: ' . $total .
             ' (' . $totalIva . ' IVA incluido)</div>';
        echo '</div>';
    }
}
```

Ahora vamos a desarrollar la clase Producto, que va a tener esas cuatro propiedades y esos cuatro métodos. **$cantidad** contiene cuantos elementos de ese producto estoy comprando (por defecto 1). Recordemos que esta clase contiene los elementos del carrito y no el contenido del almacén, que podría ser otra clase que no vamos a implementar.

```
class Producto
{
   private $nombre;
   private $precio;
   private $iva;
   private $cantidad = 1;

   public function mostrar(){
   }
   public function precio(){
   }
   public function precioIva(){
   }
   public function __construct($nombre, $precio, $iva=21){
   }
}
```

```
class Producto
{
    private $nombre;
    private $precio;
    private $iva;
    private $cantidad = 1;

    public function mostrar(){
        return "<p>{$this->nombre}: $this->precio &euro; + {$this-
>iva}%</p>";
    }

    public function precio(){
        return $this->precio;
    }

    public function precioIva(){
        return round($this->precio + ($this->precio * $this->iva /
100), 2);
    }

    public function __construct($nombre, $precio, $iva=21){
        $this->nombre = $nombre;
        $this->precio = $precio;
        $this->iva = $iva;
    }
}
```

El archivo **index.php** podría ser el siguiente:

```
<?php
include "carrito.php";
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Carrito de la compra</title>
    <link rel="stylesheet" href="estilo.css">
</head>
<body>
    <?php
    $p1 = new Producto("Espuma de afeitar", 3.5);
    $p2 = new Producto("Cereales bolas de chocolate", 5.99);
    $p3 = new Producto("Servilletas 20x20", 1.2);

    $carrito = new Carrito();
    $carrito->meter($p1);
    $carrito->meter($p2);
    $carrito->meter($p3);

   $carrito->mostrar();
   ?>
</body>
</html>
```

Versión 2 del Carrito de la compra

Ahora pasamos a la versión 2.0 de nuestro carrito.

- Vamos a añadir el método quitar() a nuestra clase Carrito para poder eliminar productos del carrito. De momento, por sencillez, lo haremos basándonos en el índice del producto dentro del array.

- También vamos a gestionar el tema de las unidades de los productos pudiendo añadir o eliminar unidades a un producto que ya esté en nuestro carrito. Para ello nos basaremos en nuevos métodos a implementar en la clase Producto.

En la clase carrito añadimos:

```
    public function quitar($indice){

    }

    public function masUnidad($indice){

    }
    public function menosUnidad($indice){

    }
```

Y en la clase Producto:

```
    public function masUnidad($unidades = 1){

    }

    public function menosUnidad(){

    }
```

En la clase carrito modificamos y añadimos los siguientes:

```
    public function precio(){
       return $this->precio * $this->cantidad;
    }

    public function precioIva(){
       return round($this->cantidad * $this->precio * (1 + $this-
    >iva / 100), 2);}

    public function mostrar(){
       return "<p><span>({$this->cantidad}x)</span> {$this->nombre}:
    $this->precio &euro; + {$this->iva}%</p>";
    }

    public function quitar($indice){
       unset($this->productos[$indice]);
    }

    public function masUnidad($indice){
       $this->productos[$indice]->masUnidad();
    }

    public function menosUnidad($indice){
       $this->productos[$indice]->menosUnidad();
    }
```

Y en la clase Producto añadimos los métodos que nos permiten añadir o suprimir unidades del producto seleccionado. Una posible mejora que no hemos implementado en la clase carrito es que cuando decrementamos el número de unidades de un producto, si éste llega a cero, podemos eliminarlo del carrito de forma automática. Se deja como ampliación.

```
    public function masUnidad($unidades = 1){
       $this->cantidad += $unidades;
    }
    public function menosUnidad(){
       if($this->cantidad > 0){
          $this->cantidad --;
       }
    }
```

El archivo **index.php** podría ser el siguiente:

```
<?php
include "carrito.php";
include "producto.php";
?>
<!DOCTYPE html>
…
<body>
    <?php
    $p1 = new Producto("Espuma de afeitar", 10);
    $p2 = new Producto("Cereales bolas de chocolate", 5.99);
    $p3 = new Producto("Servilletas 20x20", 1.2);

    $carrito = new Carrito();
    $carrito->meter($p1);
    $carrito->meter($p2);
    $carrito->meter($p3);
    $carrito->quitar(1);

    $carrito->masUnidad(0);
    $carrito->masUnidad(0);
    $carrito->menosUnidad(0);
    $carrito->menosUnidad(2);
    $carrito->menosUnidad(2);

   $carrito->mostrar();
   ?>
</body>
</html>
```

Versión 3 del Carrito de la Compra

Ahora vamos a evolucionar nuestro carrito para añadir el polimorfismo que inicialmente nos pedían en el enunciado, y lo vamos a hacer a través de una interfaz.

La interfaz, llamada iEnCarrito, contendrá los métodos comunes a todas las clases cuyos objetos puedan estar en el carrito. Por ejemplo, son comunes los métodos mostrar(), precio() y precioIva(), así como masUnidad() y menosUnidad() y todos aquellos que veamos iguales en todas las clases.

Ahora, tanto la clase Producto, como las nuevas Descuento y Pack van a implementar esta interfaz, lo cual implica que tienen que declarar todos esos métodos, aunque en el caso de los descuentos no se permitan acumular.

```
interface iEnCarrito{
    public function mostrar();
    public function precio();
    public function precioIva();
    public function masUnidad();
    public function menosUnidad();
    public function permiteUnidades();
}
```

La evolución de la clase Producto es:

```
class Producto implements iEnCarrito
{
     private $nombre;
     private $precio;
     private $iva;
     private $cantidad = 1;

     public function permiteUnidades(){
          return true;
     }
     public function mostrar(){
          return "<p><span>({$this->cantidad}x)</span> {$this->nombre}: $this->precio &euro; +
{$this->iva}%</p>";
     }
     public function precio(){
          return $this->precio * $this->cantidad;
     }
     public function precioIva(){
          return round($this->cantidad * $this->precio * (1 + $this->iva / 100), 2);
     }
     public function masUnidad($unidades = 1){
          $this->cantidad += $unidades;
     }
     public function menosUnidad(){
          if($this->cantidad > 0){
               $this->cantidad --;
          }
     }
     public function __construct($nombre, $precio, $iva=21){
          $this->nombre = $nombre;
          $this->precio = $precio;
          $this->iva = $iva;
     }
}
```

La nueva clase descuento sería:

```
class Descuento implements iEnCarrito{
    private $motivo;
    private $descuento;

    public function mostrar(){
        return "<p class=\"descuento\">$this->motivo $this->descuento
&euro;</p>";
    }
    public function precio(){
        return -$this->descuento;
    }
    public function precioIva(){
        return $this->precio();
    }
    public function masUnidad(){
        return false;
    }
    public function menosUnidad(){
        return false;
    }
    public function permiteUnidades(){
        return false;
    }
    function __construct($motivo, $descuento){
        $this->motivo = $motivo;
        $this->descuento = $descuento;
    }
}
```

Y la clase Pack es:

```
class Pack implements iEnCarrito{
    private $productosPack;
    private $cantidad = 1;

    function __construct($arrayProductos){
        $this->productosPack = $arrayProductos;
    }

    public function mostrar(){
    $salida = '<div class="pack">';
    foreach ($this->productosPack as $key => $producto) {
        $salida .= $producto->mostrar();
        $salida .= "<br />";
    }
    $salida .= '</div>';
    return $salida;
    }
    public function permiteUnidades(){
        return true;
    }

    public function precio(){
        $total = 0;
        foreach ($this->productosPack as $key => $producto) {
            $total += $producto->precio();
        }
        return $total * $this->cantidad;
    }
```

```
    public function precioIva(){
        $total = 0;
        foreach ($this->productosPack as $key => $producto) {
            $total += $producto->precioIva();
        }
        return $total * $this->cantidad;
    }

    public function masUnidad($unidades = 1){
        $this->cantidad += $unidades;
    }

    public function menosUnidad(){
        if($this->cantidad > 0){
            $this->cantidad --;
        }
    }
}
```

Y en la clase Carrito modificamos los siguientes métodos:

```
public function meter(iEnCarrito $elemento){
    $this->productos[] = $elemento;
}

public function mostrar(){
    $total = 0;
    $totalIva = 0;
    echo '<div class="carrito">';
    foreach ($this->productos as $key => $prod) {
        echo '<article class="lineacarrito">';
        echo $prod->mostrar();

        $enlaceMasUnidad = "?accion=masunidad&indice=$key";
        $enlaceMenosUnidad = "?accion=menosunidad&indice=$key";
        $enlaceEliminar = "?accion=eliminar&indice=$key";
        if($prod->permiteUnidades()){
            echo "<a href=\"$enlaceMenosUnidad\">-</a> / <a href=
\"$enlaceMasUnidad\">+</a>";
        }
        echo "<a class=\"enlaceeliminar\" href=\"$enlaceEliminar\">Eliminar</a>";
        echo '</article>';
        $total += $prod->precio();
        $totalIva += $prod->precioIva();
    }
    echo '<div class="totalcarrito">TOTAL: ' . $total . ' (' . $totalIva . ' IVA
incluido)</div>';
    echo '</div>';
    }
```

Y el nuevo archivo index.php es:

```php
<?php
include "carrito3.php";
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Carrito de la compra</title>
    <link rel="stylesheet" href="estilo.css">
</head>
<body>
    <?php
    $p1 = new Producto("Espuma de afeitar", 10);
    $p2 = new Producto("Cereales bolas de chocolate", 5.99);
    $p3 = new Producto("Servilletas 20x20", 1.2);

    $carrito = new Carrito();
    $carrito->meter($p1);
    $carrito->meter($p2);
    $carrito->meter($p3);
    $carrito->quitar(1);
```

```
$carrito->masUnidad(0);
$carrito->masUnidad(0);
$carrito->menosUnidad(0);
$carrito->menosUnidad(2);
$carrito->menosUnidad(2);

$d1 = new Descuento("Código XDDS12233", 2);

$carrito->meter($d1);

$p5 = new Producto("Cámara canon x2", 96);
$p6 = new Producto("Tarjeta de memoria 8Gb", 12);
$p7 = new Producto("Mini trípode", 5);
$pack1 = new Pack(array($p5, $p6, $p7));
$carrito->meter($pack1);

$carrito->mostrar();

?>
</body>
</html>
```

Ejercicio

El nuevo método mostrar de la clase Carrito nos muestra unos enlaces que permiten variar el número de unidades de los productos del carrito y la posibilidad de eliminarlos.

Pero no están programados. Aprovechando que no hemos visto en el ejemplo el uso de sesiones para guardar los datos del carrito, podemos plantear la mejora correspondiente.

Así pues, queda para vosotros el uso de sesiones en el Carrito de la Compra. Así, cuando pinchemos en los enlaces creados, recibiremos por GET los valores para modificar nuestro carrito, lo cual hemos de implementar y volver a presentar.