

# Parte VIII

# Traits



# Traits

Los traits permiten solventar las carencias de la herencia simple, pudiendo usar en las clases código de uno o varios traits.

Conceptualmente son estructuras parecidas a las interfaces, ya que lo que escribamos en un trait lo podemos usar en las clases aunque éstas pertenezcan a jerarquías de herencia muy distintas.

Pero en posibilidades superan a las interfaces y se aproximan a las clases, pues permiten tener propiedades, tener métodos totalmente implementados con código, usar propiedades y métodos estáticos, etc.

Por tanto, es una manera de poder hacer herencia múltiple. Es decir, que una clase disponga de métodos y propiedades de dos traits (no clases) de jerarquías distintas. Por ejemplo, la clase PerroCompañía puede heredar de Perro y de Mascota.



# Síntaxis de un trait

```
trait EnviarCopiaEmail
{
    private $to = "iescierva.carlos@gmail.com";
    private $asunto = "Copia de Seguridad";

    private function estado() {
        //Cosas que queremos hacer
        return serialize($this);
    }

    public enviar() {
        mail($this->to, $this->asunto, $this->estado());
    }
}
```



# Usar un trait

Para que una clase pueda usar aquellos traits que necesite:

```
class Blog
{
    use EnviarCopiaEmail, ConsumidorPlantilla;

    //Resto de código de la clase
}
```

Como podemos ver, la sintaxis es utilizando la palabra reservada **use**. Y podemos usar tantos traits como queramos, simplemente separados por comas. O también utilizando varios **use** en líneas diferentes.

Por tanto, los traits los podemos considerar como contenedores de código que queremos reutilizar en clases totalmente distintas entre sí.



# Resolver Conflictos

Si dos traits insertan un método con el mismo nombre, se produce un error fatal, siempre y cuando no se haya resuelto el conflicto.

Para resolver los conflictos de nombres entre Traits en una misma clase se debe usar el operador `insteadof` (en lugar de) para elegir unívocamente uno de los dos.

Como esto sólo permite excluir métodos, se puede usar el operador `as` para permitir incluir el método excluido con otro nombre.

A continuación mostramos un ejemplo. La primera clase usa el método `reír` de B excluyendo el de A y el método `llorar` al revés.

En la segunda clase, queremos usar los dos métodos `reír`, por lo que elegimos el método del Trait B y renombramos el de A.



# Resolver Conflictos

```
trait A {  
    public function reir() {  
        echo "Me rio con A";  
    }  
  
    public function llorar() {  
        echo "Lloro con A";  
    }  
}
```

```
trait B {  
    public function reir() {  
        echo "Me rio con B";  
    }  
  
    public function llorar() {  
        echo "Lloro con B";  
    }  
}
```

```
class ReirLlorar  
{  
    use A, B {  
        B::reir insteadof A;  
        A::llorar insteadof B;  
    }  
    //Código de la clase  
}
```

```
class ReirLlorar2  
{  
    use A, B {  
        B::reir insteadof A;  
        A::llorar insteadof B;  
        A::reir as reirMucho;  
    }  
    //Código de la clase  
}
```



# Ejemplo Carríto de la compra

Vamos a retomar el ejemplo del carríto de la compra que ya vimos con detalle y lo evolucionaremos, aplicando el uso de traits, de autocarga y ya puestos, resolveremos el ejercicio planteado dándole significado a los enlaces '+', '-' y 'eliminar'.

En primer lugar, vamos a separar cada una de las clases en un archivo distinto, los cuales (por comodidad) los guardaremos en la misma carpeta raíz donde se encuentra index.php. Cada archivo se llamará NombreDeLaClase.php.



# Paso 1: Uso de autocarga

Ahora, el archivo `index.php`, que empezaba con código PHP para poder incluir la clase `Carrito`, en lugar de escribir varios `includes` con todas las clases, vamos a usar autocarga.

Por tanto, comienza de la siguiente manera, sustituyendo la línea que incluía la clase `Carrito` por:

```
<?php
spl_autoload_register(function($clase) {
    $archivo = $clase.'.php';
    include $archivo;
});
?>
```

que como vemos hacemos uso de la versión con función anónima que explicamos.



# Paso 2: Uso de traits

Vemos que en las clases `Producto` y `Pack` tenemos dos métodos con el mismo código. Como siempre, **repetir código no es una buena práctica**.

Para mejorar, vamos a quitar los métodos `masUnidad()` y `menosUnidad()` de las dos clases, y los vamos a utilizar para definir un trait, que ambas usarán. Además también tienen la propiedad `$cantidad` que se maneja en ambos métodos. Por tanto esa propiedad también la eliminamos de ambas clases, para hacerla aparecer en el trait.

Usamos un trait y no una clase padre porque las clases `Producto` y `Pack` no tienen ninguna relación entre ambas.



Creamos el archivo MasMenos.php con el siguiente código:

```
<?php
trait MasMenos{

    private $cantidad = 1;

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

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

Y en las clases Producto y Pack eliminamos esos dos métodos y la propiedad, añadiendo como primera línea de ambas clases

```
use MasMenos;
```

que como ya hemos visto, implica que todo lo que hay en el trait está disponible para dichas clases.



# Paso 3: Utilizando los enlaces

Para poder utilizar los enlaces hemos de tener en cuenta que cuando pinchemos en cualquiera de ellos, nos llevará a otra página o a la misma (nosotros elegiremos la misma). Pero eso implica acabar el script `index.php` para volverlo a ejecutar.

Y sabemos que cuando un script se acaba todas sus variables desaparecen (son eliminadas de la memoria).

Como nosotros queremos que el objeto carrito (con todos los datos que lleva) permanezca, la mejor forma es usar sesiones.

Si queremos que de una visita a otra (por ejemplo en dos días distintos) también se mantenga, una alternativa es usar cookies, o también serializar el objeto y guardarlo en la BD junto con nuestros datos de usuario.



# Paso 3 continuación

En primer lugar, vamos a modificar nuestro archivo index.php para que admita sesiones.

Para ello, antes del comienzo del HTML, debajo de la función de autocarga añadiremos el uso de sesiones.

Queda así:

```
<?php
spl_autoload_register(function($clase) {
    $archivo = $clase.'.php';
    include $archivo;
});
session_start();
?>
<!DOCTYPE html>
...
```



# Paso 3 continuación

Y al final del archivo añadimos un enlace con la opción de eliminar la sesión.

```
?>
```

```
<br><br>
```

```
<p><a href="destroy.php">Eliminar sesión</a></p>
```

```
</body>
```

```
</html>
```

que nos lleva a un archivo destroy.php con el código:

```
<?php
```

```
spl_autoload_register(function($clase) {
```

```
    $archivo = $clase.'.php';
```

```
    include $archivo;
```

```
});
```

```
session_start();
```

```
session_destroy();
```

```
?>
```

```
<a href=".">volver</a>
```



# Paso 4: Un único carrito

El archivo `index.php` tiene la siguiente línea:

```
$carrito = new Carrito();
```

que es la que nos permite crear el carrito.

Pero resulta que nuestra operativa es la siguiente:

- Si el usuario entra por primera vez a la página debemos crear un carrito.
- Si no es la primera vez que accede (porque p.e. haya pulsado los enlaces) no se debe crear un nuevo carrito, sino utilizar el que ya existe.

Por tanto, queremos que cada usuario sólo pueda tener a la vez un único objeto de la clase `Carrito`.



## Paso 4 continuación

La solución que vamos a plantear es añadir a la clase Carrito un constructor vacío (sin código) y definido como privado. Con ello conseguimos que desde fuera de la clase no se pueda crear un objeto de dicha clase (al ser privado, desde fuera no podemos acceder al constructor y por tanto el objeto no se construye). Así pues, la línea comentada del index hemos de suprimirla para que no genere un error.

Añadimos a la clase Carrito el siguiente método:

```
private function __construct() {  
}
```

Ahora bien, si con ese método no podemos instanciar un objeto de la clase con la sentencia **new Carrito**,

¿Cómo lo hacemos?



## Paso 4 continuación

Vamos a crear un método estático en la clase Carrito (el cual, por ser estático, está disponible sin necesidad de crear ningún objeto) que lo que va a hacer es comprobar si tenemos guardado en la sesión un carrito o no. Si ya tenemos guardado uno, nos devuelve ese. Si no, crea uno nuevo.

```
public static function getCarrito() {  
    if(isset($_SESSION["micarrito"])) {  
        return $_SESSION["micarrito"];  
    }  
    $carrito = new Carrito();  
    $_SESSION["micarrito"] = $carrito;  
    return $carrito;  
}
```

Y en el index, en lugar de la línea suprimida escribimos:

```
$carrito = Carrito::getCarrito();
```



# Paso 5: Listado de Productos

En este paso, vamos a mostrar debajo del carrito un listado con los productos que tenemos para poder comprar y un enlace para poder comprarlos. La realidad nos dice que esto tendría que ir en otra página, pero por no complicarlo más, los vamos a mostrar aquí.

El enlace nos tiene que llevar a nuestra propia página index, indicando que es lo que queremos hacer. Vamos a seguir la misma dinámica que con los enlaces de `masUnidad` y `menosUnidad` indicando que la acción es comprar y en lugar de pasarle el id de producto (que no lo tenemos) le enviamos el propio objeto serializado.

Para ello, como ya tenemos `mostrar()` en la clase `Producto` que sirve para otra cosa, vamos a aprovechar que hemos visto los métodos mágicos y usaremos `__toString()`.

Este método se ejecuta cuando tratamos de imprimir un objeto directamente.



# Paso 5 continuación

Añadimos a la clase Producto el método

```
public function __toString() {  
    $salida = "<br>" . $this->nombre . " " . $this->precio .  
        " &euro;";  
    $salida .= " <a href=\"?accion=comprar&producto=\"" .  
        urlencode(serialize($this)) . "\">Comprar</a>";  
    return $salida;  
}
```

Y en el index.php, después de mostrar el carrito, añadimos las siguientes líneas de código:

```
echo "<br><br>";  
echo $p1;  
echo $p2;  
echo $p3;
```



## Paso 5 continuación

Ahora, si pinchamos en comprar sucede lo mismo que cuando pinchamos en '+' o '-' en los productos del carrito, NADA.

Y no hace nada porque las acciones que están definidas no las tenemos todavía codificadas.

Una aclaración: Como estamos en un ejemplo, debe quedar claro que el objetivo es practicar lo que hemos visto. Pero no lo estamos haciendo del todo bien. No es adecuado que vía GET mandemos el objeto serializado. Lo correcto sería tener un formulario que nos permitiera enviarlo por POST y además, los productos deberían estar almacenados en una BD.

No es que no sepamos hacerlo, es que nos llevaría días. Por eso, en este caso prima el desarrollo didáctico del ejercicio sobre lo demás.



# Paso 6: Programar los Enlaces

Ahora vamos a dar sentido a los enlaces que tenemos, que son cuatro. En la línea de producto del carrito tenemos '+', '-' y 'eliminar' y debajo del carrito el que acabamos de crear en el paso anterior para cada producto, 'comprar'.

Cada vez que pulsemos un enlace, la página `index.php` se recarga, lo cual significa (por lo hecho en los pasos 2 y 3) que se ejecuta el método `getCarrito()`.

Desde ese método llamaremos a otro, `operaciones()`, que vamos a crear y que va a llevar toda la codificación de los enlaces.

Así, cuando devolvamos un nuevo carrito o el existente, se habrá visto modificado por el enlace que hayamos pulsado.



# Paso 6 continuación

El método `operaciones()` que vamos a añadir a la clase `Carrito` es:

```
private function operaciones() {  
    if(isset($_GET["accion"])) {  
        if($_GET["accion"]=="comprar") {  
            $elem = unserialize($_GET["producto"]);  
            $this->meter($elem);  
        }  
        if($_GET["accion"]=="eliminar") {  
            $this->quitar($_GET["indice"]);  
        }  
        if($_GET["accion"]=="menosunidad") {  
            $this->menosUnidad($_GET["indice"]);  
        }  
        if($_GET["accion"]=="masunidad") {  
            $this->masUnidad($_GET["indice"]);  
        }  
    }  
}
```



## Paso 6 continuación

El método `operaciones()` no está perfectamente hecho, ya que sólo comprobamos la existencia de `$_GET["accion"]`, pero no del resto. Queda para vosotros esa mejora.

El método `getCarrito()` hay que modificarlo, quedando así:

```
public static function getCarrito() {  
    if(isset($_SESSION["micarrito"])) {  
        $carrito = $_SESSION["micarrito"];  
    } else {  
        $carrito = new Carrito();  
        $_SESSION["micarrito"] = $carrito;  
    }  
    $carrito->operaciones();  
    return $carrito;  
}
```

Ahora los enlaces ya son funcionales, aunque no perfectos.



## Paso 6 continuación

Y no son perfectos porque si pinchamos dos veces en comprar el mismo artículo, lo añade dos veces, en lugar de incrementar en una unidad la primera existencia del mismo.

Igualmente, si pulsamos el '-' repetidamente hasta llegar a '0' no se elimina del carrito.

Ambas mejoras las dejo para que vosotros las desarrolléis.



# Paso 7: Listado de Descuentos y Packs

Igual que mostramos los productos debajo del carrito para que se puedan comprar, vamos a ampliar para que también se muestren los descuentos y los pack. La operativa a seguir es la misma.

Añadimos el siguiente método a la clase Descuento:

```
public function __toString() {  
    $salida = "<br>" . $this->motivo . " " . $this->descuento  
    . " &euro;";  
    $salida .= " <a href=\"?accion=comprar&producto=\"" .  
urlencode(serialize($this)) . "\">Comprar</a>";  
    return $salida;  
}
```



# Paso 7 continuación

Y este otro a la clase Pack:

```
public function __toString() {  
    $salida = "<br>Pack: " . $this->precio() . " &euro;";  
    $salida .= " <a href=\"?accion=comprar&producto=" .  
urlencode(serialize($this)) . "\">Comprar</a>";  
    return $salida;  
}
```

Y en el index.php añadimos:

```
echo "<br><br>";  
echo $p1;  
echo $p2;  
echo $p3;  
echo $d1;  
echo $pack1;
```



# Paso 7 continuación

Vemos que dado que el Pack no tiene un atributo nombre, sólo mostramos 'Pack', lo cual no es indicativo de lo que vamos a comprar.

Queda para vosotros la siguiente mejora: En lugar de aparecer la cadena 'Pack', debería aparecer un listado de los productos que componen el pack, para que la persona que quiera comprarlo sepa qué es.



# Paso 8: Optimizando Código

Una vez que hemos desarrollado los métodos `__toString()` en las tres clases, nos debemos de dar cuenta que la línea donde indicamos el enlace

```
" <a href=\"?accion=comprar&producto=\" .  
urlencode(serialize($this)) . "\">Comprar</a>"
```

es la misma en los tres. Y hemos comentado que no es aconsejable repetir código, porque el mantenimiento se puede llegar a volver insostenible.

Debemos crear un método común a las tres clases que contenga ese código, y la única manera de hacerlo es creando un `trait`, puesto que las interfaces no llevan código.



## Paso 8 continuación

Creamos el archivo EnlaceComprar.php con el código:

```
<?php
trait EnlaceComprar{
    private function enlaceComprar() {
        return " <a href=\"?accion=comprar&producto=" .
urlencode(serialize($this)) . "\">Comprar</a>";
    }
}
```

En los tres métodos `__toString()` cambiamos la línea indicada por esta otra:

```
$salida .= $this->enlaceComprar();
```

Y en las tres clases añadimos el nuevo trait:

```
use EnlaceComprar;
```



# Paso 9: Persistencia

Ahora queremos persistencia en nuestro carrito. De momento, si salimos del navegador y volvemos a entrar, la sesión se cierra y por tanto el carrito se pierde.

Lo que queremos es que no se pierda. Es decir, que persista entre sesiones. Porque puede que estemos comprando, nos tengamos que ir, y pasado un tiempo, querer seguir con la compra, allí donde la dejamos.

Para conseguir nuestro objetivo, hemos de hacer uso de las cookies, que a falta de BD, es la mejor manera de lograr la persistencia.



## Paso 9 continuación

Como lo que queremos hacer es guardar en una cookie un objeto (en nuestro caso el Carrito), en lugar de definir los métodos nuevos en la clase Carrito, lo vamos a hacer en un trait que luego añadiremos.

Y elegimos un trait, porque cuando tengamos otros proyectos (o ampliamos este) que necesiten guardar un objeto en una cookie el código es el mismo y lo podremos reutilizar fácilmente.

Ese trait, que vamos a llamar Persistir, contendrá dos métodos. Uno para guardar el estado del carrito en una cookie y otro para recuperarlo desde la cookie. Además, contendrá un atributo donde definiremos el tiempo de vida de la cookie.



# Paso 9 continuación

Creamos el archivo Persistir.php con el código:

```
<?php
trait Persistir{
    private $segundosPersistencia = 2592000;

    public function guardaEstadoCookie($cookie) {
        $tiempo = time()+$this->segundosPersistencia;
        setcookie($cookie, serialize($this), $tiempo);
    }
    public static function traeCookie($nombrecookie) {
        if(isset($_COOKIE[$nombrecookie])) {
            return unserialize($_COOKIE[$nombrecookie]);
        }
        return false;
    }
}
```



## Paso 9 continuación

Una vez creado el trait, tenemos que añadir la línea

**use Persistir;**

en la clase Carrito, para que así podamos utilizar sus métodos.

El método **getCarrito()** hemos de adaptarlo para que no sólo consulte si tenemos un carrito en la sesión, sino también consulte si tenemos uno en una cookie. Por eso, el método **traeCookie()** lo hemos definido como estático, ya que tenemos que usarlo sin haber creado un objeto de dicha clase. En caso de que no haya ningún carrito en la sesión ni en una cookie crearemos uno nuevo.


Además, también hemos de modificar el archivo `destroy.php`, para que antes de destruir la sesión, guardemos el estado del carrito en la cookie.



# Paso 9 continuación

El método `getCarrito()` quedaría como

```
public static function getCarrito() {  
    if(isset($_SESSION["micarrito"])) {  
        $carrito = $_SESSION["micarrito"];  
    } elseif($carrito = self::traeCookie("carrito")) {  
        $_SESSION["micarrito"] = $carrito;  
    } else {  
        $carrito = new Carrito();  
        $_SESSION["micarrito"] = $carrito;  
    }  
    $carrito->operaciones();  
    return $carrito;  
}
```





# Paso 9 continuación

Y en el archivo `Destroy.php` añadimos dos líneas nuevas

```
<?php
spl_autoload_register(function($clase) {
    $archivo = $clase.'.php';
    include $archivo;
});
session_start();
$carrito = Carrito::getCarrito(); ←
$carrito->guardaEstadoCookie("carrito"); ←
session_destroy();

?>
<a href=".">volver</a>
```



## Paso 9 continuación

Pero, con lo que acabamos de hacer no es suficiente todavía, ya que si cerramos el navegador y lo volvemos a abrir, nos aparece el carrito vacío.

Es debido a que la información relativa a las cookies se envía en la cabecera del script, antes de que la pagina reciba código HTML. Luego con lo que tenemos no estamos haciendo un uso adecuado de las cookies.

Para arreglar este problema y conseguir total persistencia, vamos a añadir una línea nueva en la cabecera de index.php

```
<?php
spl_autoload_register(function($clase){...});
session_start();
$carrito = Carrito::getCarrito(); ←
?>
<!DOCTYPE html>
...
```



## Paso 9 continuación

Con la línea añadida podemos recuperar el estado del carrito aunque se cierre el navegador, ya que se está recuperando de la cookie.

¿Pero donde lo guardamos en la cookie?

Es conveniente guardar el carrito en la cookie cada vez que se actualice. Por eso, volvemos a modificar el método `getCarrito()` de la clase `Carrito` añadiendo la siguiente línea:

```
public static function getCarrito() {  
    ...  
    $carrito->operaciones();  
    $carrito->guardaEstadoCookie("carrito");  
    return $carrito;  
}
```

