# TEMA 4. GENERACIÓN DINÁMICA DE PÁGINAS WEB.

## Objetivos

- Entender cómo construir una aplicación web dinámica a través de mecanismos de separación entre la lógica de negocio y la presentación de la página.
- Analizar y distinguir qué significa una arquitectura de capas y una arquitectura de niveles.
- Conocer que son los patrones de software y como estos pueden facilitar la construcción de un sitio web.
- Estudiar las herramientas para la generación dinámica de páginas web, tanto en entorno cliente como servidor.

## Contenidos

Podemos separar el diseño de la página de la lógica del negocio a la hora de construir la página web. Esto permite por ejemplo que en la edición digital de un diario exista una persona encargada de los contenidos que será el periodista, una persona encargada de la organización y presentación de los contenidos que será el diseñador y por último el programador que se encargará de la funcionalidad de la página y de que todos sus elementos se ejecuten correctamente. Una herramienta en la que se percibe claramente las separación de roles son los gestores de contenidos o CMS como Wordpress, Joomla, Prestashop, etc...

### 4.1.-. Mecanismos de separación de la lógica de negocio.

#### a) Modelos Físicos de separación: Arquitectura Multinivel.

El modelo físico está formado por los elementos hardware que constituyen el sistema.
Actualmente se utiliza la arquitectura multinivel que deriva de la arquitectura cliente-servidor.
Cada elemento del sistema con un rol distinto represente un nivel diferente. Normalmente nos encontramos con sistemas de dos o más niveles.

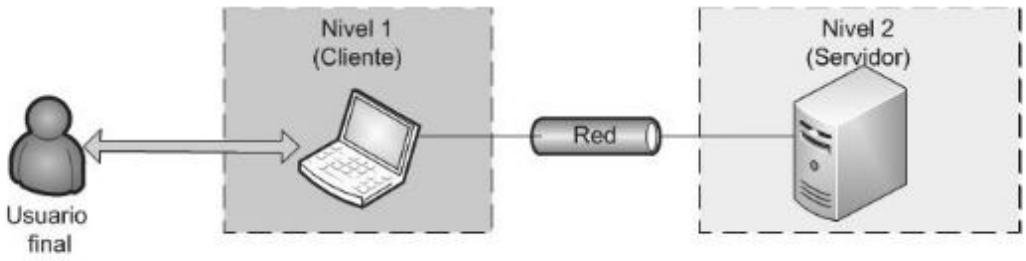



*Figura 4.2. Ejemplo de arquitectura de 2 niveles*

Normalmente en arquitecturas de más de dos niveles, los nuevos elementos desdoblan las funciones del servidor, apareciendo un nuevo servidor que descarga la funcionalidad y la carga de trabajo al servidor único.

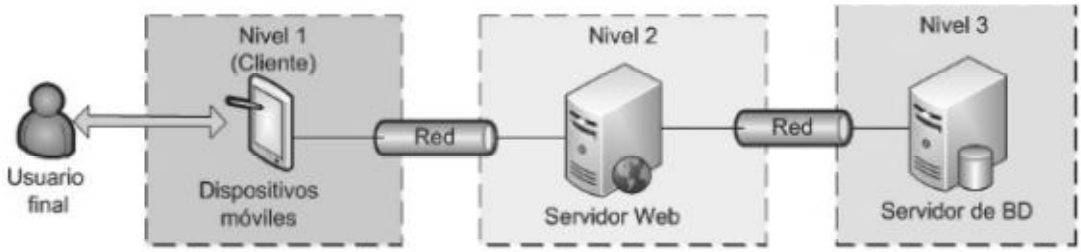



*Figura 4.3. Ejemplo de arquitectura a tres niveles*

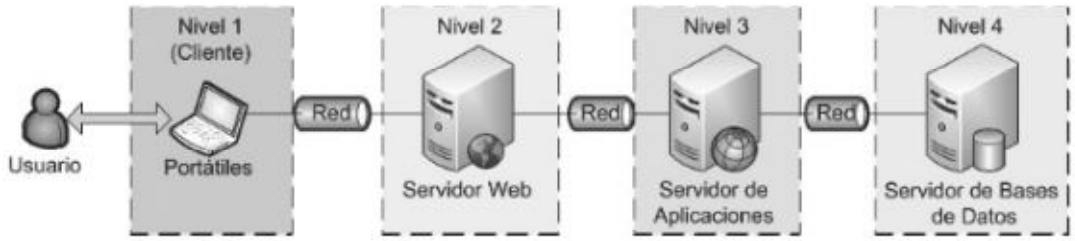



*Figura 4.4. Arquitectura estándar para la construcción de páginas web*

### b) Modelos de separación lógicos.

Hace referencia a la manera en que dividimos o distribuimos los componentes software para obtener un mejor rendimiento del sistema.

Uno de los mecanismos más utilizados a la hora de construir una web es la arquitectura de capas que divide el desarrollo de la aplicación en varios niveles lógicos o formas de organizar el código.

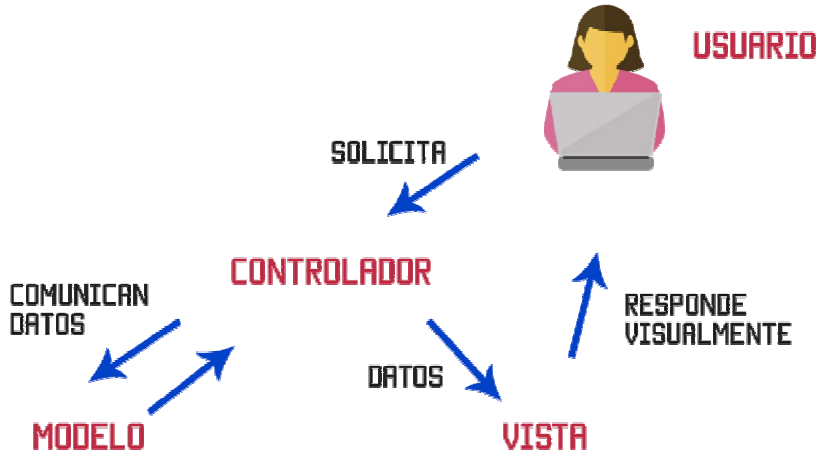
Las principales ventajas del desarrollo de aplicaciones mediante la arquitectura de capas son:

- Facilidad de reusabilidad.
- Facilidad de mantenimiento.
- Desarrollo de capas paralelo.
- Aplicaciones más robustas gracias al encapsulamiento.
- Reducción de costos.

### Arquitectura de capas Modelo-Vista-Controlador.

El patrón arquitectónico con esquema Modelo-Vista-Controlador (MVC) busca separar la aplicación en tres componentes principales: el modelo, la vista y el controlador:

- **Modelo.** Se encarga de los datos, generalmente consultando la base de datos (altas, bajas, modificaciones, consultas, búsquedas, listados, ..)
- **Vista**. Son la representación visual de los datos, todo lo que tenga que ver con la interfaz gráfica va aquí. Ni el modelo ni el controlador se preocupan de cómo se verán los datos, esa responsabilidad es únicamente de la vista.
- **Controlador.** Se encarga de recibir las órdenes del usuario, solicitar los datos al modelo y de comunicárselos a la vista.




El modelo es el responsable de:

- Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.
- Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser: "Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor".
- Lleva un registro de las vistas y controladores del sistema.
- Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero por lotes que actualiza los datos, un temporizador que desencadena una inserción, etc.).

El controlador es responsable de:

- Recibe los eventos de entrada (un clic, un cambio en un campo de texto, etc.).
- Contiene reglas de gestión de eventos, del tipo "SI Evento Z, entonces Acción W". Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método "Actualizar()". Una petición al modelo puede ser "Obtener_tiempo_de_entrega ( nueva_orden_de_venta )".

Las vistas son responsables de:

- Recibir datos del modelo y los muestra al usuario.
- Tienen un registro de su controlador asociado (normalmente porque además lo instancia).
- Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

Ejemplo del esquema MVC: Tenemos un sistema para borrar productos. Cuando se hace una solicitud a una página para borrar un producto de la base de datos, se pone en marcha un controlador que recibe el identificador del producto que se tiene que borrar. Entonces le pide al modelo que lo borre y a continuación se comprueba si el modelo nos responde que se ha podido borrar o no. En caso que se haya borrado queremos mostrar una vista y en caso que no se haya borrado queremos mostrar otra.

### Sentencias include y require.

Las sentencias include y require permiten incluir en una página web, el código escrito en otro fichero. Las características y el modo de uso de include y de require son identicas, salvo por el mensaje de error generado si falta un documento. Con include, si el nombre de archivo no existe, se recibe una advertencia (warning) y el script continua su ejecución. En cambio, con require el script se detendrá devolverá un error fatal.

La sintaxis es la siguiente:

```
include('fichero.php');

require('fichero.php');
```

Por tanto, require es idéntico a include excepto que en caso de fallo, require producirá un error fatal de nivel E_COMPILE_ERROR (detiene el script) mientras que include sólo emitirá una advertencia (E_WARNING) lo cual permite continuar el script.

La sentencia require_once es idéntica a require excepto que PHP verificará si el archivo ya ha sido incluido y si es así, no se incluye de nuevo.

En la sentencia include_once si el código del fichero ya ha sido incluido, no se volverá a incluir.

**Ejemplo:**

Archivo del Controlador: **controlador.php**

```
<?php
      // Incluir la lógica del modelo
      include('modelo.php');

      // Obtener los datos de un artículo
      $cod_articulo = 2;
      $articulo = getArticulo($articulos, $cod_articulo);

      // Incluir la lógica de la vista
      include('vista.php');
?>
```

Archivo del Modelo: **modelo.php**

```
<?php
      $articulos = array(array(100, "Monitor Samsung V-43e", 134.56),
                         array(230, "Portatil Asus A-512p", 621.33),
                         array(324, "Proyector Epson RF500", 339.77),
                         array(451, "Router TPlink G540", 84.20));
      function getArticulo($articulos, $cod){
            $datos_articulo = $articulos[$cod];
            return $datos_articulo;
      }
?>
```

Archivo del vista: **vista.php**

```
<html>
  <head>
    <title>Consulta de Articulo</title>
  </head>
  <body>
    <h1>Datos del Articulo</h1>
    Código: <?php echo $articulo[0]; ?><br>
    Descripcion: <?php echo $articulo[1]; ?><br>
    Precio: <?php echo number_format($articulo[2], 2, ",", ".")." €"; ?>
  </body>
</html>
```

### c) Arquitectura Orientada a Servicios (SOA).

La Arquitectura Orientada a Servicios SOA es vista como un conjunto de servicios que intercambian mensajes XML utilizando protocolos de transporte como HTTP. Los Servicios Web establecen un mensaje común mediante el cual distintos sistemas pueden comunicarse entre sí, facilitando la construcción de sistemas distribuidos heterogéneos.

## 4.2.-. Mecanismos de generación dinámica de interfaces web.

Una página dinámica es aquella que se crea en tiempo de ejecución y puede presentar diferente información para cada usuario que esté visualizando la página en función de las necesidades de cada usuario.

Hay tres tipos de sitios web dinámicos:

- El dinamismo se encuentra en el lado del cliente.
- El dinamismo se centra en el lado del servidor.
- El dinamismo se divide entre el cliente y el servidor.

### a) Creación de contenido dinámico del lado del cliente.

La base de la creación de páginas web se realiza con el lenguaje HTML, pero este lenguaje no fue diseñado para la creación de páginas dinámicas, por lo que se han diseñado una serie de tecnologías que sirven para crear contenidos dinámicos.

Algunos de las tecnologías y lenguajes script del lado del cliente son HTML Dinámico, Active X, Applets y plugins específicos para contenidos dinámicos:

- **Javascript** es un lenguaje de programación que se ejecuta del lado del servidor y permite interactuar con la página creando contenidos dinámicos. Las sentencias javascript se insertan entre las etiquetas <script></script>.
- **Applets**. Son pequeñas aplicaciones que realizan tareas especificas. Se incrustan dentro de la página y están realizados en Java. Se pueden ejecutar en cualquier navegador que tenga instalada la Máquina Virtual de Java. Cuando se utiliza un applet se descarga desde el servidor el fichero que debe ejecutar la máquina virtual que se denomina bytecode.
- **Active X**. Este tipo de controles fueron desarrollados por Microsoft para dar respuesta a la tecnología Java. Son programas precompilados por lo que su ejecución es más eficiente.

- **Plugins**. Los plugins son componentes que permiten mejorar y ampliar las funciones que realiza la página. Por ejemplo, el plugin para mostrar documentos PDF o animaciones Flash. Para poder utilizar los plugins debemos instalar previamente la aplicación correspondiente en el equipo cliente.

### b) Creación de contenido dinámico del lado del servidor.

Las páginas dinámicas del lado del servidor personalizan su respuesta de acuerdo a la petición y los datos proporcionados por el cliente. Los datos son pasados al servidor a través de los métodos POST o GET.

Algunos de las tecnologías y lenguajes script del lado del cliente son:

- **CGI (Common Gateway Interface): CGI: Interfaz de Pasarela Común.** Es un Estándar para transferir datos entre el cliente y un programa residente en el servidor. Permite al servidor web interactuar con aplicaciones para realizar consultas de bd, documentos, generadores de emails, comercio electrónico y procesar solicitudes... Suelen estar instalados en un subdirectorio específico /cgi-bin. Perl: lenguaje de programación para construir CGIs.
- **Servlets.** Programa escrito en Java que se ejecuta en el servidor, y no en el navegador como los applets.
- **Lenguajes de Script como PHP, JSP, ASP.** Creados para genera páginas web dinámicas introduciendo código interpretado. Se debe instalar en el servidor web el interpretador de lenguaje script usado y se intercalan fragmentos de código script dentro de un documento HTML. El código HTML es la parte estática y el código script es la parte dinámica. Ejemplos de lenguaje script de servidor: PHP, JSP, ASP. ASP.NET

### c) Creación de contenido dinámico del lado del cliente y del servidor.

Incorporan tecnologías vistas en los apartados anteriores, tanto del lado del cliente como del lado del servidor.

## 4.3.-. Introducción a la Programación Orientada a Objetos en PHP.

### 4.3.1.- Programación con objetos.

En la programación orientada a objetos en primer lugar debemos crear una clase de objeto. Al crear la clase definiremos su propiedades que son las características del objeto y sus métodos que son las funciones que puede realizar el objeto. Por último crearemos los objetos concretos de esa clase que son las instancias del objeto.

### Creación de clases

Para definir una clase de objetos, utilizaremos la palabra reservada **class** seguida del nombre de la clase. A continuación, entre llaves escribiremos las propiedades y los métodos de la clase.

Ejemplo:

```
class Alumno {
      :     :     :
}
```

### Definir propiedades

Una vez creada la clase, dentro de las llaves escribiremos las propiedades que los objetos de esta clase deben tener utilizando variables.

Ejemplo:

```
class Alumno {
	var $dni;
	var $nombre;
	var $apellidos;
	var $curso;
	var $nota_teoria;
	var $nota_practica;
	var $nota_final;
}
```

Si inicializamos la propiedad esta tendrá ese valor por defecto.

Podemos guardar el código del objeto en un archivo php con nombre **clase_nombreClase.php**. Cuando lo necesitemos será llamado en a la página mediante la instrucción include().

Otra posibilidad es utilizar la palabra reservada public o private para indicar el tipo acceso a los atributos del objeto.

Ejemplo:

```
class Alumno {
	private $dni;
	private $nombre;
	private $apellidos;
	private $curso;
	private $nota_teoria;
	private $nota_practica;
	private $nota_final;
}
```

### Constructor de objeto.

Cuando instanciamos una clase automáticamente se ejecuta el constructor de la clase que normalmente inicializa los atributos del objeto con los datos pasados como parámetros al constructor.

Para crear una función constructora debemos llamarla de la misma manera que la clase a la que pertenece, por ejemplo Alumno.

```
function nombre_clase($argum1, $argum2, ... $argumN) {
```

```
        :     :     :     :     :     :     :
    }
```

La función constructora puede llevar o no argumentos, y dentro de las llaves podemos acceder las propiedades del objeto mediante la variable $this .

Para llamar a una función constructora lo haremos en el momento de instanciar el objeto:

```
$instancia_objeto = new nombre_clase($arg1,arg2, ... argN);
```

Es decir, al instanciar el objeto, le pasamos los argumentos, de manera que desde el mismo momento en que se crea el objeto, este puede tener valores en sus propiedades.
Al pasar los argumentos debemos de tener cuidado de pasarlos en el mismo orden en que están en la función constructora para que cada uno de ellos se corresponda con la propiedad a la que corresponde.

Ejemplo:

```
function Alumno($d, $n, $a, $c, $nT, $nP){
      $this->dni = $d;
      $this->nombre = $n;
      $this->apellidos = $a;
      $this->curso = $c;
      $this->nota_teoria = $nT;
      $this->nota_practica = $nP;
}
```

Otra posibilidad es crear el constructor creamos una función con el nombre reservado **__construct().**

Ejemplo:

```
class Alumno {
      private $dni;
      private $nombre;
      private $apellidos;
      private $curso;
      private $nota_teoria;
      private $nota_practica;
      private $nota_final;

      function __construct($dni, $nombre, $apellidos, $curso,
                  $nota_teoria, $nota_practica, $nota_final){
            $this->dni=$dni;
```

```
            $this->nombre=$nombre;
            $this->apellidos=$apellidos;
            $this->curso=$curso;
            $this->nota_teoria=$nota_teoria;
            $this->nota_practica = $nota_practica;
            $this->nota_final = $nota_final;
        }
    }
```

### Instanciar un objeto

Se llama instanciar un objeto al hecho de crear un objeto concreto que pertenezca a una clase.

Ejemplo:

```
$alumno_1005 = new Alumno('23.535.590-T', 'Luis', 'Sanz Oliver',
                    '1DAW', 7.9, 5.4, 0);
```

Es decir para instanciar un objeto de una clase lo definimos en una variable ($alumno_1005) y como valor le asignamos la palabra reservada new seguido del nombre de la clase a la que va a pertenecer y entre paréntesis los parámetros pasados al constructor.

### Acceso a las propiedades del objeto

Podemos asignar valor a una propiedad del objeto utilizando el nombre de la instancia del objeto, los símbolos -> y el nombre de la propiedad seguido de un igual y el valor que queremos asignarle, siempre y cuando la propiedad sea **public.**

Ejemplo:

```
$alumno_1005 -> nombre = "Mario";
```

En caso de atributos private utilizaremos métodos **set** para asignar valores a los atributos.

Podemos mostrar por pantalla el valor de una propiedad de una instancia de un objeto utilizando el nombre de la instancia del objeto, los símbolos -> y el nombre de la propiedad, siempre y cuando la propiedad sea **public**.

Ejemplo:

```
echo $alumno_1005 -> nombre;
```

En caso de atributos private utilizaremos métodos **get** para recuperar valores a los atributos.

## 4.3.2.- Métodos en objetos

Los métodos representan acciones que puede realizar el objeto. Para definir un método, utilizamos la palabra reservada **function** seguida del nombre del método y entre paréntesis los parámetros que le pasamos. Podemos añadir delante de la palabra reservada function el tipo de acceso permitido para la función (public o private).

Dentro de la función utilizaremos la variable **$this** para hacer referencia al propio objeto, así como la palabra reservada **return** seguida de un valor para que el método devuelva un valor.

Para llamar a un método de un objeto utilizaremos las sintaxis:

```
$nombre_objeto->nombre_metodo()
```

Podremos acceder al método desde fuera de la clase siempre y cuando no se trate de un método de tipo private.

Por ejemplo en la clase alumno podemos crear un método que calcule la nota final de un alumno a partir de la nota de teoría y de prácticas y devuelva true si el alumno a aprobado o false en caso contrario. Al método se le pasarán como parámetros el porcentaje que representa la nota de teoría y la nota de prácticas para el cálculo de la nota final. Asignaremos en el método las sentencias para añadir el valor a la propiedad nota_final.

Ejemplo:

```
public function calculaNotaFinal($porcTeoria, $porcPractica) {
      $notaFinal = ($this->nota_teoria*$porcTeoria)
                  + ($this->nota_practica*$porcPractica);
      $this->nota_final = $notaFinal;
      if ($notaFinal >= 5)
            $aprobado = true;
      else
            $aprobado = false;
      return $aprobado;
}
```

En la página principal llamaremos al método para que haga el cálculo de la nota final del alumno.

Ejemplo:

```
$alumno_1005->calculaNotaFinal(0.3, 0.7);
```

o también podemos obtener el valor que devuelve la función e indica si el alumno está aprobado o suspendido.

Ejemplo:

```
$aprobado_1005 = $alumno_1005->calculaNotaFinal(0.20, 0.80);
echo ($aprobado_1005)?"Aprobado<br>":"Suspendido<br>";
```

Podemos crear un método que nos muestre por pantalla las propiedades del objeto. Con una simple llamada al método, podremos mostrar los datos del objeto.

Ejemplo:

```
public function toString() {
      echo "DNI: ".$this->dni."<br>";
      echo "Nombre: ".$this->nombre."<br>";
      echo "Apellidos: ".$this->apellidos."<br>";
      echo "Curso: ".$this->curso."<br>";
      echo "Nota Teoría: ".$this->nota_teoria."<br>";
      echo "Nota Práctica: ".$this->nota_practica."<br>";
      echo "Nota final: ".$this->nota_final."<br>";
}
```

A continuación se muestra el código completo de la clase Alumno llamado **clase_alumno.php** y de la página principal **index.php.**

**Fichero: clase_alumno.php**

```
<?php
      class Alumno {
            private $dni;
            private $nombre;
            private $apellidos;
            private $curso;
            private $nota_teoria;
            private $nota_practica;
            private $nota_final;

            public function __construct($dni, $nombre, $apellidos,
                  $curso, $nota_teoria, $nota_practica, $nota_final){
                  $this->dni=$dni;
                  $this->nombre=$nombre;
                  $this->apellidos=$apellidos;
                  $this->curso=$curso;
                  $this->nota_teoria=$nota_teoria;
                  $this->nota_practica = $nota_practica;
                  $this->nota_final = $nota_final;
            }

            public function mostrarAlumno(){
                  echo "$this->dni<br>";
                  echo "$this->nombre<br>";
                  echo "$this->apellidos<br>";
                  echo "$this->curso<br>";
                  echo "$this->nota_teoria<br>";
                  echo "$this->nota_practica<br>";
                  echo "$this->nota_final<br>";
```

```
            }

            public function calculaNotaFinal($porcTeoria,
             $porcPractica) {
                  $notaFinal = ($this->nota_teoria*$porcTeoria)
                  + ($this->nota_practica*$porcPractica);
                  $this->nota_final = $notaFinal;
                  if ($notaFinal >= 5)
                        $aprobado = true;
                  else
                        $aprobado = false;
                  return $aprobado;
            }
      }
?>
```

**Fichero: index.php**

```
<?php
      include("clase_alumno.php");
      $alumno = new Alumno("25.354.986-R", "Luis", "Ros Más",
            "Desarrollo Web en Entorno Servidor", 7.2, 5.6, 0);
      $aprobado = $alumno->calculaNotaFinal(0.3, 0.7);
      $alumno->mostrarAlumno();
      echo ($aprobado)?"Aprobado<br>":"Suspendido<br>";
?>
```

### 4.3.4.- Subclases

Las subclases son una serie de objetos que derivan de una clase principal. Estas subclases se denominan también clases extendidas, o clases derivadas.

Al crear estas subclases tenemos que hacer referencia a la clase de la que dependen mediante la palabra reservada extends.

```
class nombre_subclase extends nombre_clase {
      :     :     :     :     :
 }
```

Por ejemplo

```
class Universitario extends Alumno {
      :     :     :     :     :
 }

class Graduado extends Alumno {
      :     :     :     :     :
 }
```

Las subclases heredan las propiedades y métodos que tiene la clase principal de la que derivan, lo que se denomina herencia.

Las clases Universitario y Graduado tienen las mismas propiedades y métodos que la clase Alumno. Si la clase Alumno tiene función constructora, ésta también se hereda a las subclases.

```
$instancia_subclase = nombre_subclase($arg1,$arg2,$arg3, ... ,$argN)
```

Los argumentos ($arg1,$arg2,$arg3, ... ,$argN) hacen referencia a la función constructora de la clase principal.

La clase principal define por tanto una serie de propiedades y métodos generales que tendrán todas las subclases. Las propiedades y métodos particulares de cada subclase los definiremos dentro de las llaves que delimitan el código de la subclase.

Nada nos impide crear más subclases dentro de las subclases que tenemos, de manera que se cree una jerarquía de clases, en la cual cada una hereda todos los métodos y propiedades de las que deriva.

El polimorfismo es una cualidad por la cual podemos redefinir las propiedades y métodos que una subclase ha heredado, de manera que el cambio afecte sólo a esa subclase.

Por ejemplo, podemos redefinir el método mostrar() de la clase principal, que muestra en pantalla las propiedades comunes a todas las subclases, para mostrar también las propiedades específicas de la subclase.

Podemos redefinir en una subclase un método de la clase principal llamando en la subclase al mismo método de la clase principal y posteriormente añadir el código especifico necesario de la subclase, evitando tener que reescribir nuevamente el código del método de la clase principal.

```
parent::metodo_clase();
```

Del mismo modo, la función constructora de la clase principal, sólo nos inicializa las propiedades comunes, pero no las específicas, por lo tanto debemos crear una función constructora para las subclases, que incluya el código de la función constructora de la clase principal, y además el código específico para inicializar las propiedades específicas de la subclase.

La función constructora lleva el nombre de la subclase a la que pertenece, y además debe llevar todos los argumentos que hay que pasarle, tanto de las propiedades comunes como de las específicas, éstos últimos se ponen a continuación de los comunes.

Llamamos a la función constructora principal mediante la palabra reservada **`parent::`**, tal como hacíamos con los otros métodos, la función constructora se llama como en la clase

principal, ya que es ésta a la que llamamos. Por otra parte indicamos también aquí los argumentos de las propiedades comunes, que se pasan a través del constructor principal.

Podemos declarar una clase principal como abstracta, lo que quiere decir que no puede ser instanciada, es decir no podremos crear objetos a partir de ellas. Además puede incorporar métodos abstractos. Los métodos abstractos son aquellos que solo existe su declaración, dejando su implementación a las clases extendidas o derivadas.

Todos los métodos declarados como abstractos, deberán pertenecer necesariamente a una clase abstracta. Es decir que una clase normal no podremos definir un método como abstracto.

Una clase abstracta tiene la misma estructura que una clase normal pero es necesario añadir la palabra clave **`abstract`** al inicio de su declaración.

```
abstract class nombre_clase_principal {
:     :     :     :     :
}
```

**Ejemplo:** Tenemos una clase principal Alumno que no puede ser instanciada, de la que derivan las subclases Universitario y Graduado con la diferencia que los Graduados no hacen examen práctico sino que realizan Actividades Prácticas a lo largo de la evaluación y que pueden ser Aptas o No aptas. La nota final de los Graduados será 0 si no han realizado las actividades prácticas o la nota teórica en caso de que las Actividades sean Aptas. Los universitarios tiene examen teórico y práctico y cada uno de ellos representan un porcentaje de la nota final que se pasa por parámetro al método que calcula la nota final del alumno.

**Fichero: abstract_Alumno.php**

```php
<?php
  abstract class Alumno {
    private $dni;
    private $nombre;
    private $apellidos;
    private $curso;
    private $nota_final;

    public function __construct($dni, $nombre, $apellidos, $curso){
      $this->dni = $dni;
      $this->nombre = $nombre;
      $this->apellidos = $apellidos;
      $this->curso = $curso;
    }

    public function mostrarAlumno() {
      echo "DNI: ".$this->dni."<br>";
      echo "Nombre: ".$this->nombre."<br>";
```

```
      echo "Apellidos: ".$this->apellidos."<br>";
      echo "Curso: ".$this->curso."<br>";
    }
  }


  class Universitario extends Alumno {
    private $nota_teoria;
    private $nota_practica;

    public function __construct($dni, $nombre, $apellidos, $curso,
$nota_teoria, $nota_practica){
      parent::__construct($dni, $nombre, $apellidos, $curso);
      $this->nota_teoria = $nota_teoria;
      $this->nota_practica = $nota_practica;
    }

    public function mostrarUniversitario() {
      parent::mostrarAlumno();
      echo "Nota Teoría: ".$this->nota_teoria."<br>";
      echo "Nota Práctica: ".$this->nota_practica."<br>";
    }

    public function calculaNotaFinal($porcTeoria, $porcPractica) {
      $notaFinal = ($this->nota_teoria*$porcTeoria)
        + ($this->nota_practica*$porcPractica);
      $this->nota_final = $notaFinal;
    }
  }

  class Graduado extends Alumno {
    private $eval_actividades;
    private $nota_teoria;

    public function Graduado($dni, $nombre, $apellidos, $curso,
$eval_actividades, $nota_teoria){
      parent::__construct($dni, $nombre, $apellidos, $curso);
      $this->eval_actividades = $eval_actividades;
      $this->nota_teoria = $nota_teoria;
    }

    public function mostrarGraduado() {
      parent::mostrarAlumno();
      echo "Evaluación Actividades: ".$this->eval_actividades."<br>";
      echo "Nota Teoría: ".$this->nota_teoria."<br>";
    }
```

```
    public function calculaNotaFinal() {
      if ($this->eval_actividades == "S")
        $this->nota_final = $this->nota_teoria;
      else
        $this->nota_final = 0;
    }
  }
?>
```

**Fichero: index.php**

```
<?php
  include("abstract_Alumno.php");

  $universitario_0172= new Universitario("25.354.986-R", "Luis",
    "Ros Más", "Desarrollo Web en Entorno Servidor", 7.2, 5.6);
  $graduado_0491= new Graduado("15.865.346-P", "Marta",
    "Amoros Solís", "Desarrollo Web en Entorno Cliente", "S", 7.4);
  $graduado_0582= new Graduado("16.622.198-H", "Sara","Serrano Mir",
      "Programación", "N", 5.2);

  $universitario_0172->calculaNotaFinal(0.3, 0.7);
  $universitario_0172->mostrarUniversitario();
  echo str_repeat("*",20)."<br>";

  $graduado_0491->calculaNotaFinal();
  $graduado_0491->mostrarGraduado();
  echo str_repeat("*",20)."<br>";

  $graduado_0582->calculaNotaFinal();
  $graduado_0582->mostrarGraduado();
?>
```