

Parte 3

Aproximación a la POO en JavaScript

Capítulo 3-1

Enfoque orientado a “objetos” en JavaScript

1. Introducción

Aunque la implementación del modelo de programación orientada a objetos (POO) no esté tan completa en JavaScript como en C++ o Java, JavaScript ofrece los mecanismos principales gestionados por estos lenguajes.

Recordemos los conceptos más importantes de la POO:

- Encapsulación: reunión de un conjunto de propiedades (parte de tratamiento de datos) y de funciones, también llamadas métodos (parte de procesamientos), dentro de un objeto tipo (quizás es más correcto hablar de clase), con la posibilidad de crear (instanciar) objetos a partir de esta clase.
- Herencia: posibilidad de "fabricar" una nueva clase a partir de una clase existente; esta nueva clase hereda las propiedades y métodos de la clase padre (se pueden añadir nuevas propiedades/métodos a la nueva clase).
- Polimorfismo: un método del mismo nombre asociado a varias clases puede tener comportamientos diferentes para algunas de estas clases.

2. Programación orientada a objetos a través de ejemplos

JavaScript siempre ha sido una pieza esencial en los desarrollos web, principalmente para la programación del lado "cliente", es decir, del lado del navegador. Habitualmente, sin sumergirse de lleno en el lenguaje, los desarrolladores producen código JavaScript de calidad mediocre, contentándose con adaptar el código fuente recuperado de sitios web y manipulando los conceptos POO lo menos posible.

En paralelo, han aparecido un gran número de librerías JavaScript y su uso permite producir aplicaciones de mejor calidad. El dominio de estas librerías supone tener conocimientos básicos de POO en JavaScript.

Por tanto, el objetivo de la exposición que sigue es presentarle lo que hay que saber sobre este asunto. Los conceptos se van a explicar a través de una serie de ejemplos.

Algunos lectores que ya tengan una importante experiencia en otros lenguajes POO (PHP 5, Java, C++...) al principio se pueden sentir incómodos con los aspectos específicos de la POO en JavaScript, la POO por prototipado.

2.1 Secuencia 1: Declaración de los objetos JavaScript de manera "Inline"

Se trata de la manera más sencilla de declarar un objeto en JavaScript.

```
/* Declaración inline de un objeto JavaScript */
/* NB: Esta técnica no permite la herencia a partir del objeto
más adelante */
var Adicion = {
    x: 5,
    y: 10,
    calculo: function()
    {
        return this.x + this.y;
    }
};

/* Uso del objeto Adicion */
document.write("Suma: " + Adicion.calculo());
```

El resultado obtenido de la ejecución será el siguiente:

```
■ Suma = 15
```

En este tipo de declaración de objeto, es posible prever la especificación de atributos (propiedades) y también de métodos.

La palabra clave **this** sirve para indicar que se está haciendo referencia a los atributos del objeto en sí mismo.

Está claro que con este tipo de declaración no será posible reutilizar este tipo de definición para crear un objeto de las mismas características (o parecidas). Por tanto, este método se utilizará poco (o nada) porque no permite la herencia. No se preocupe, porque volveremos sobre esto más en detalle, a lo largo de esta exposición.

2.2 Secuencia 2: Creación de objetos JavaScript con un constructor

También es posible crear nuestros objetos JavaScript con un constructor (concepto bien conocido en los lenguajes POO). En JavaScript, será suficiente con escribir una función y llamarla posteriormente con la palabra clave **new**. Por tanto, la función jugará el papel de clase sin serlo realmente.

Veamos con un ejemplo el desarrollo que es necesario seguir:

```
/* Definición de una función constructor, de nombre Coche */
var Coche = function()
{
    /* Atributo(s) del objeto */
    this.tieneMotor = true;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
        document.write("avanza");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Visualización del atributo tieneMotor del objeto simca1100 */
if (simca1100.tieneMotor)
{
    document.write("El coche simca1100 tiene un motor<br />");
}
else
{
    document.write("El coche simca1100 no tiene motor !<br />");
}

/* Llamada al método avanzar del objeto simca1100 */
document.write("El coche simca1100 ");
simca1100.avanzar();
```

En nuestro ejemplo, se define en primer lugar una función **Coche**. Integra un atributo booleano que indica que los coches tienen un motor y un método (función) de nombre **avanzar**, que mostrará "**avanza**" cuando se pida, a partir de un objeto de tipo **Coche** (se entenderá rápidamente).

Posteriormente, se construye un objeto de nombre **simca1100** a partir del constructor **Coche** (siento que el término "constructor" pueda ser confuso en un ejemplo basado en coches):

```
/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();
```

Ahora, la propiedad (atributo) **tieneMotor** se puede consultar para el objeto **simca1100** instanciado y también se puede ejecutar el método **avanzar**. Cuando se ejecute, tendremos:

```
El coche simca1100 tiene un motor
El coche simca1100 avanza
```

2.3 Secuencia 3: Variables privadas en una instancia de objeto

En el ejemplo de la secuencia anterior, habrá observado que las propiedades (atributos) llevan como prefijo la palabra clave **this**. Esto es lo que las hace usables desde el exterior (caso de la propiedad **tieneMotor**). Por el contrario, por necesidades locales del constructor (cálculo interno), puede declarar variables no expuestas, usando como prefijo la palabra clave **var**.

Veamos un ejemplo concreto:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function()
{
    /* Variable(s) local(s) no accesible(s) desde el exterior del objeto */
    var numeroRuedas = 4;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
        document.write("avanza");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método avanzar del objeto simca1100 */
document.write("El coche simca1100 ");
simca1100.avanzar();

/* Intento de visualización de la variable local del constructor Coche */
document.write("<br />");
document.write("El coche simca1100 tiene " + simca1100.numeroRuedas +
" ruedas");
```

Se usa una variable local **numeroRuedas** en el constructor con el prefijo **var**. Esto solo es accesible, como estaba previsto, desde dentro del constructor, como se muestra en la ejecución de este script:

```
El coche simca1100 avanza
El coche simca1100 tiene undefined ruedas
```

2.4 Secuencia 4: Paso de argumento(s) a un constructor

En el ejemplo siguiente, vamos ver que es posible pasar uno o varios argumentos (lo habíamos visto ya para las funciones clásicas) a un constructor:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function(modelo)
{
    /* Atributo(s) del objeto informado durante la instanciación */
    this.modelo = modelo;
}

/* Instanciación de un objeto simca1100 a través del constructor Coche
con paso de argumento */
var miCoche = new Coche("simca1100");

/* Visualización del atributo modelo del objeto miCoche */
document.write("Tengo un " + miCoche.modelo);
```

El argumento **modelo** está en los paréntesis que siguen al nombre del constructor:

```
var Coche = function(modelo)
```

y está disponible en el cuerpo del constructor por:

```
    this.modelo = modelo;
```

A continuación, es suficiente a nivel de la instanciación del objeto **miCoche** con pasar como argumento un valor ("simca1100" en nuestro caso):

```
var miCoche = new Coche("simca1100");
```

La propiedad (atributo) **modelo** del objeto se mostrará por:

```
document.write("Tengo un " + miCoche.modelo);
```

2.5 Secuencia 5: No compartición de los métodos por las instancias de objetos

Dado que los métodos se declaran durante la instanciación de los objetos, sus definiciones están duplicadas en memoria.

En un ejemplo pequeño, el impacto es bajo.

Por el contrario, si su aplicación manipula muchos objetos con métodos múltiples y complejos en los constructores, esto se convierte en inmanejable.

El siguiente ejemplo destaca el problema que acabamos de comentar:

```
/* Definición de una función constructor de nombre Coche */
var Coche = function()
{
    /* Atributo(s) del objeto */
    this.tieneMotor = true;
    /* Método(s) del objeto */
    this.avanzar = function()
    {
        document.write("avanza");
    }
    this.retroceder = function()
    {
        document.write("retrocede");
    }
}

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Instanciación de un objeto renault12 a través del constructor Coche */
var renault12 = new Coche();

/* Comprobación de la igualdad de métodos avanzar de los objetos simca1100
y renault12 */
if (simca1100.avanzar == renault12.avanzar)
{
    document.write("Método avanzar compartido por los objetos simca1100 y
Renault12<br />");
}
else
{
    document.write("Método avanzar no compartido por los objetos simca1100
y Renault12<br />");
}
```

La ejecución confirma que el método **avanzar** no está factorizado:

■ Método avanzar no compartido por los objetos simca1100 y Renault12

La noción de prototipo que vamos a descubrir a continuación va a resolver este problema.

2.6 Secuencia 6: Noción de prototipo

Un prototipo es un conjunto de elementos (atributos/propiedades y métodos) que se va a asociar a un constructor (sin "almacenamiento" en el constructor en sí mismo). Durante la ejecución, cuando una propiedad de objeto solicitada en el código no se encuentre en el constructor del objeto en cuestión, se realizará una búsqueda en esta lista "adicional".

Veamos un ejemplo completo:

```
/* Definición de una función constructor de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Comprobación de la existencia por defecto de un prototipo para
cualquier constructor */
document.write("Prototipo del constructor: " + Coche.prototype);

/* Añadir un método zigzaguar al prototipo del constructor Coche */
Coche.prototype.zigzaguar = function()
{
    document.write("zigzaguea peligrosamente<br />");
};

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método zigzaguar del objeto simca1100, accesible a través
del prototipo del constructor Coche */
document.write("<br />¿Qué hace el simca1100? ");
simca1100.zigzaguar();

/* Instanciación de un objeto renault12 a través del constructor Coche */
var renault12 = new Coche();

/* Comprobación del método zigzaguar compartido o no, entre los objetos
simca1100 y renault12 */
if (simca1100.zigzaguar == renault12.zigzaguar)
{
    document.write("Método zigzaguar compartido por los objetos
simca1100 y renault12<br />");
}
else
{
    document.write("Método zigzaguar no compartido por los objetos
simca1100 y renault12<br />");
}
```

La opción que se ha utilizado en el ejemplo es codificar un constructor vacío. Después, se realiza una comprobación para demostrar que cualquier constructor tiene un prototipo:

```
document.write("Prototipo del constructor: " + Coche.prototype);
```

Posteriormente, se ha asociado un método **zigzaguitar** al prototipo relacionado con el constructor **Coche**:

```
Coche.prototype.zigzaguitar = function()  
{  
    document.write("zigzaguita peligrosamente<br />");  
};
```

Después de la instanciación habitual de un objeto **simca1100** a partir del constructor **Coche**, se hace una llamada al método **zigzaguitar** para el objeto **simca1100**. También se instancia un segundo objeto, **renault12**, a partir del constructor **Coche** para demostrar que esta vez el método **zigzaguitar** está compartido (es común a ambos objetos).

La ejecución da:

```
Prototipo del constructor: [object Object]  
¿Qué hace el simca1100? Zigzaguita peligrosamente  
Método zigzaguitar compartido por los objetos simca1100 y renault12
```

2.7 Secuencia 7: Sobrecarga de un método

Para una instancia dada de objeto, es posible sobrecargar (modificar) un método (o una propiedad/atributo).

Veamos un ejemplo concreto de cómo aplicarlo:

```
/* Definición de una función constructor, de nombre Coche */  
/* NB: El constructor aquí está vacío */  
var Coche = function() {};  
  
/* Añadir un método cargar al prototipo del constructor Coche */  
Coche.prototype.cargar = function()  
{  
    document.write("carga<br />");  
};  
  
/* Instanciación de un objeto simca1100 a través del constructor Coche */  
var simca1100 = new Coche();  
  
/* Llamada al método cargar del objeto simca1100, accesible a través  
del prototipo del constructor Coche */  
document.write("El simca1100 ");
```

```
simca1100.cargar();

/* Instanciación de un objeto renault12 a través del constructor Coche */
var renault12 = new Coche();

/* Modificación (sobrecarga) del método cargar para el objeto
simca1100 */
simca1100.cargar = function()
{
    document.write("carga rápidamente <br />");
};

/* Llamada al método cargar (sobrecargado) del objeto simca1100 */
document.write("El simca1100 ");
simca1100.cargar();

/* Llamada al método cargar (no sobrecargado) del objeto renault12 */
document.write("El renault12 ");
renault12.cargar();
```

La ejecución da como resultado:

```
El simca1100 carga
El simca1100 carga rápidamente
El renault12 carga
```

Puede comprobar que la modificación (sobrecarga) solo impacta al objeto `simca1100`.

2.8 Secuencia 8: Extensión de un prototipo

La extensión es la suma posterior de un método adicional a nivel de un prototipo de constructor. Veamos un ejemplo de extensión:

```
/* Definición de una función constructor de nombre Coche */
/* NB: El constructor aquí está vacío */
var Coche = function() {};

/* Añadir un método acelerar al prototipo del constructor Coche */
Coche.prototype.acelerar = function()
{
    document.write("acelera<br />");
};

/* Instanciación de un objeto simca1100 a través del constructor Coche */
var simca1100 = new Coche();

/* Llamada al método acelerar del objeto simca1100, accesible a través
del prototipo del constructor Coche */
```

Capítulo 2-3

Los contenedores semánticos

1. Utilizar correctamente los contenedores semánticos

Para diseñar sus páginas, debe pensar en términos de «contenedor». Un contenedor, como su nombre indica, incluye un contenido de tipo muy variado.

En estos contenedores, puede situar texto, imágenes, formularios, enlaces, tablas, etc. Pero también puede tener contenedores más «pequeños», como para resaltar una o varias palabras o para definir una celda de una tabla.

Los contenedores también sirven para estructurar sus páginas. De esta manera, puede utilizar contenedores para insertar un encabezado de página, una columna de desplazamiento (un *sidebar* en inglés), un pie de página, una barra de navegación, etc.

Como habrá entendido, todos los contenidos se insertan en contenedores. Cada contenedor, fuera de las capas (`<div>` y `<span>`), están dedicados a recibir un contenido específico. Para esto se califican los contenedores semánticos.

2. El elemento <div>

El elemento <div> es uno de los contenedores más antiguos del HTML. Permite crear una capa estructural en la página. En estas capas, podemos situar cualquier contenido, incluso otros contenedores, como otras capas <div>, párrafos, listas, etc.

El HTML5 introduce nuevos contenedores semánticos que han reducido el uso del elemento <div>. Pero el motivo por el que debe evitar usar cajas <div> no es porque utilice HTML5. Siempre se pueden usar y efectivamente tienen su utilidad. Las cajas <div> habitualmente se usan para conseguir contenedores «neutros», que no necesariamente tienen un sentido semántico concreto.

3. El elemento

El elemento se utiliza habitualmente para aplicar un formato específico a una parte del texto dentro de un párrafo. Es especialmente útil cuando se desea destacar un fragmento sin alterar la estructura del contenido.

A continuación se muestra un ejemplo concreto: queremos poner el fondo gris y un borde fino en una parte de un texto de un párrafo.

A continuación se muestra el selector CSS:

```
.fondo-gris {  
    background-color: #eee;  
    padding: 0 5px;  
    border: 1px solid #333;  
}
```

A continuación se muestra su aplicación en el código HTML:

```
<p>Donec ullamcorper nulla no metus auctor  
fringilla. Morbi leo risus, <span class="fondo-gris">porta  
ac consectetur ac vestibulum</span> at eros. Donec sed odio...</p>
```

A continuación se muestra la visualización obtenida:

Donec ullamcorper nulla non metus auctor fringilla. Morbi leo risus, porta ac consectetur ac vestibulum at eros. Donec sed odio...

4. El elemento <header>

El elemento <header> permite insertar una zona de visualización para los encabezados. Estos encabezados se pueden utilizar en varios lugares:

- A nivel de la página: es el clásico encabezado de página, habitualmente ubicado en la parte superior de la pantalla, con un logo, un eslogan, una barra de navegación principal, etc.
- A nivel de los contenidos, lo que permite tener una introducción al contenido que sigue, como el encabezado de un artículo, por ejemplo.

Este contenedor puede contener todo tipo de elementos: títulos, párrafos, enlaces, etc.

Puede anidar los elementos <header> y <footer> en otro elemento <header> si los dos primeros elementos se incluyen en un mismo elemento padre.

A continuación se muestra un ejemplo de anidación perfectamente validado:

```
<article>
  <header>
    <h2>Cras Vestibulum Sem Fermentum</h2>
    <aside>
      <header>
        <h3>Inceptos Magna Vehicula Malesuada</h3>
        <p>Mollis Risus Sollicitudin Inceptos...</p>
      </header>
      <p>Sit Mattis Aenean Commodore...</p>
      <footer>
        <p>Parturient Pharetra Quam</p>
      </footer>
    </aside>
  </header>
  <p>Adipiscing Ultricies Dapibus Mollis...</p>
</article>
```

5. El elemento `<footer>`

El elemento `<footer>` se utiliza para definir el pie de página de un documento o de una sección específica, como un artículo. Su función semántica es similar a la del elemento `<header>`, aunque no es obligatorio usar ambos conjuntamente. Es decir, puede haber un `<footer>` sin `<header>`, y viceversa.

6. El elemento `<aside>`

El elemento `<aside>` permite mostrar un contenido relacionado de manera indirecta con el contenido principal al que se le asocia. Esto se corresponde habitualmente con las clásicas barras de desplazamientos (*sidebar* en inglés), las zonas de componentes de interfaz (*widgets* en inglés), los complementos sobre los artículos o cualquier otro contenido textual.

Por lo general, el contenido principal aparece en el centro de la página, mientras que el contenido asociado al elemento `<aside>` se muestra a la derecha.

7. El elemento `<nav>`

El elemento `<nav>`, como su nombre permite intuir, se dedica a la visualización de una barra de navegación con enlaces de hipertextos. Pero atención, no se sienta obligado a tener una única zona de navegación por página. Puede crear tantos elementos `<nav>` como quiera, con navegaciones diferentes en sus páginas, siempre y cuando cada uno de ellos se identifique correctamente. El elemento `<nav>` quizás se dedica más a la navegación principal del sitio web, a la creación de enlaces entre las páginas del sitio web.

Puede incluir una navegación principal `<nav>` en un encabezado `<header>` y una navegación secundaria `<nav>` en un pie de página `<footer>`, por ejemplo.

8. El elemento `<main>`

El elemento `<main>` permite indicar el contenido principal de la página. Este contenido debe ser único y no repetirse en la página. Además, la norma indica concretamente su contexto de utilización: no se debe usar en el interior, como elemento incluido, e los elementos `<article>`, `<aside>`, `<footer>`, `<header>` o `<nav>`.

9. El elemento `<section>`

El elemento `<section>` permite agrupar los elementos que comparten una misma temática. Esto permite agrupar, en un mismo elemento, un contenido estructurado, con su encabezado y su pie de página. La utilización de varios elementos `<section>` distinguirá varias partes, varias secciones dentro de una misma página, con otros elementos de estructura anidados.

10. El elemento `<article>`

El elemento `<article>` permite insertar un contenido autónomo. Está calificado como autónomo porque se puede reutilizar en cualquier lugar en el sitio web, sin que su comprensión se vea afectada. El uso más habitual retoma el nombre del elemento: creación de artículos de blog y de actualidad.

11. El elemento `<search>`

El elemento `<search>` permite agrupar todos los elementos relacionados con una búsqueda textual en el sitio web. Puede incluir formularios, títulos, párrafos y eventualmente los resultados. Es importante mantener esta área bien estructurada para facilitar su comprensión y accesibilidad.

12. La visualización de los elementos de estructura

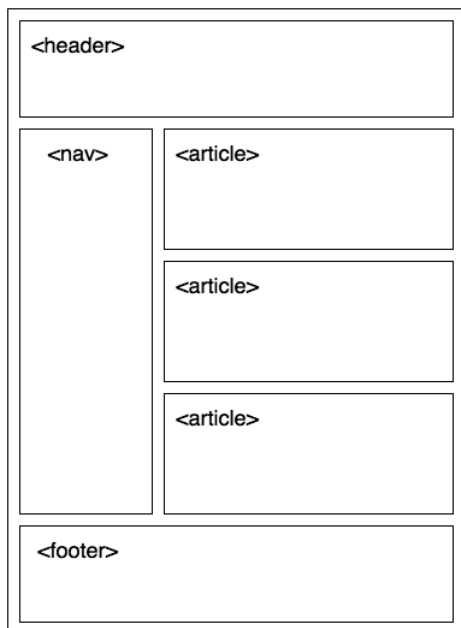
Todos los elementos de estructura de tipo block que acabamos de ver, es decir, `<div>`, `<header>`, `<footer>`, `<aside>`, `<nav>`, `<main>`, `<section>` y `<article>`, utilizan la propiedad `display`, pero no tienen un valor específico para los márgenes.

```
div, header, footer, aside, nav, main, section, article {  
    display : block ;  
}
```

13. Dos ejemplos de estructura semántica de página

13.1 Una estructura semántica sencilla

A continuación se muestra un primer ejemplo con una estructura semántica muy sencilla:

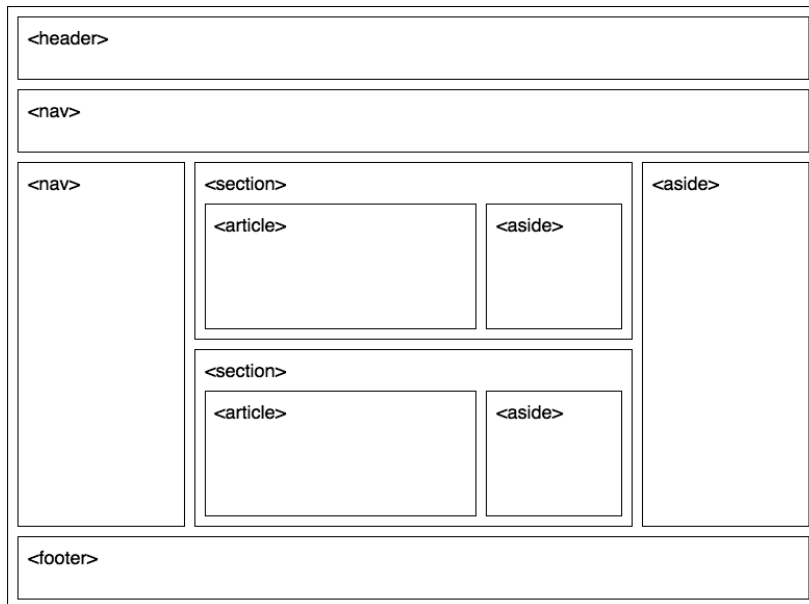


Tenemos:

- Un encabezado `<header>` en la parte superior, con un logo y un eslogan, por ejemplo.
- Una barra de navegación `<nav>` en la parte izquierda de la página.
- Toda la actualidad del sitio web se podrá situar en elementos `<article>`.
- Para terminar, el pie de página `<footer>` podrá contener las menciones legales, los enlaces de contacto, etc.

13.2 Una estructura semántica más elaborada

A continuación se muestra una segunda estructura más elaborada:



Tenemos:

- Un encabezado `<header>` en la parte superior.
- Más abajo, una barra de navegación `<nav>` para la navegación general del sitio web, para navegar entre las páginas.
- A la izquierda, una segunda caja `<nav>` para la navegación secundaria, para los enlaces relacionados directamente con el contenido de la página mostrada.
- A la derecha, un elemento `<aside>` muestra la información relacionada con el contenido de la página, como los enlaces promocionales, los contenidos relacionados, etc.
- El contenido de la página se muestra en dos elementos `<section>`, que permiten de esta manera diferenciar correctamente estos dos contenidos. Cada elemento `<section>` contiene un elemento `<article>` para el contenido textual y un elemento `<aside>` para mostrar los elementos de información adicionales relacionados con el artículo (iconografía, enlaces, etc).
- Finalmente, un pie de página `<footer>` para mostrar la información legal, las condiciones de venta, un enlace de contacto, un plano de acceso, etc.

14. Un ejemplo de estructura semántica de un artículo

A continuación se muestra un ejemplo de un artículo, de un contenido textual que utiliza estos elementos de estructura semántica:



Tenemos:

- Un elemento `<article>` como contenedor general.
- Nuestros artículos contienen encabezados, introducciones; por lo tanto utilizamos el elemento `<header>`. El elemento `<header>` contiene el título `<h1>` del artículo y su subtítulo `<h2>`.
- El contenido textual del artículo se ubica en los elementos `<p>`. El artículo contiene los enlaces a los complementos de artículos, listados en una lista `<ul>`.