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

El elemento 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 , párrafos, listas, etc.

El HTML5 introduce nuevos contenedores semánticos que han reducido el uso del elemento . Pero el motivo por el que debe evitar usar cajas no es porque utilice HTML5. Siempre se pueden usar y efectivamente tienen su utilidad. Las cajas 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.

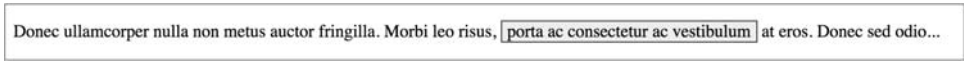
A continuación se muestra el selector CSS:

```
.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:



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 Commodo...</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

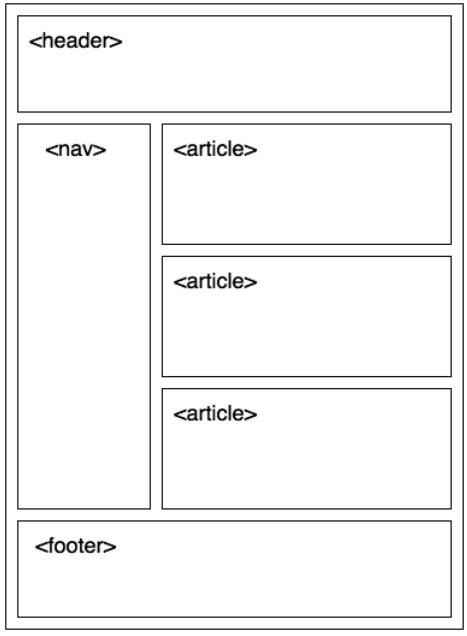
Todos los elementos de estructura de tipo `block` que acabamos de ver, es decir, `<div>`, `<header>`, `<footer>`, `<aside>`, `<nav>`, `<main>`, `<section>` y `<article>`, utilizan la propiedad `display`, pero no tienen un valor específico para los márgenes.

```
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:

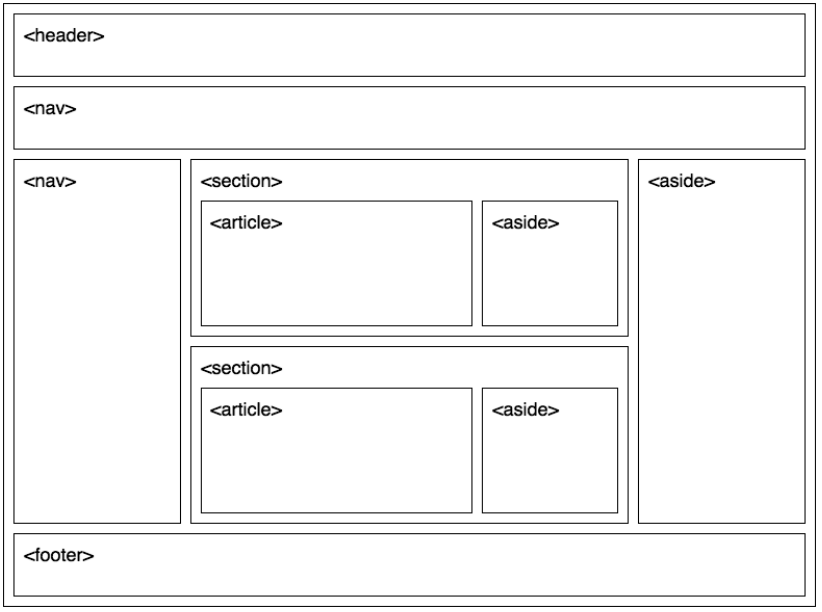


Tenemos:

– 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:

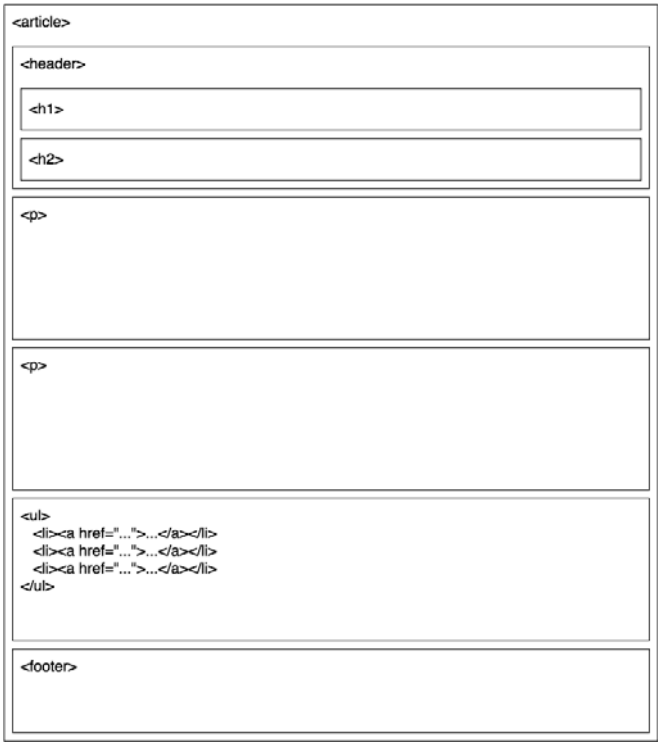


Tenemos:

– 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>`.

Capítulo 3
Programación orientada a objetos

1. Principios de la programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma muy extendido en desarrollo de software. Viene a completar un panorama que ya es muy rico del paradigma de procedimientos, así como del funcional.

La POO es una forma de diseño de código que aspira a representar los datos y las acciones como si formaran parte de clases; ellas mismas se convierten en objetos durante su creación en memoria. Este concepto se ha presentado rápidamente en el capítulo anterior: ahora es el momento de comprender su funcionamiento de manera más detallada.

1.1 ¿Qué es una clase?

Una clase es un elemento del sistema que forma la aplicación. Una clase contiene dos tipos de elementos de código: datos y métodos (que representan acciones). Hay que ver la clase como una caja donde es posible ordenar estos dos tipos de elementos. Para hacer un paralelismo con la vida real, podemos comprender fácilmente que la definición de una clase se aplica a un objeto como un ordenador, por ejemplo. Este último dispone de métodos (encender, apagar, etc.), así como de propiedades (número de pantallas, cantidad de RAM, etc.).

De manera conceptual, una clase solo es una definición. Una vez que haya decidido lo que debe contener, así como sus métodos, es conveniente crearla. Esta acción se llama instanciación. Tras esta operación, obtenemos una instancia en memoria de un objeto.

Vamos a intentar hacer una comparación. Tomemos el ejemplo de una fábrica de producción de objetos de madera. Para poder crear un objeto, se necesita un plan (la clase). Gracias a este último, la máquina puede cortar y juntar los diversos elementos (datos y métodos) para crear una instancia nueva (instanciación).

En C#, la declaración de una clase se hace con la palabra clave `class`. Hay algunas posibles particularidades, especialmente el ámbito, que estudiaremos justo después, en la sección ¿Qué se puede declarar dentro de una clase? - Métodos, así como los conceptos de `static`, `sealed` y el de `partial`. La sintaxis completa de la declaración de una clase es la siguiente:

```
AMBITO [static] [sealed] [partial] class NOMBRE_CLASE
```

El nombre de la clase es libre, pero debe cumplir dos normas:

– Solo puede contener caracteres alfanuméricos y el carácter guion bajo («_»).

– No puede empezar con un número.

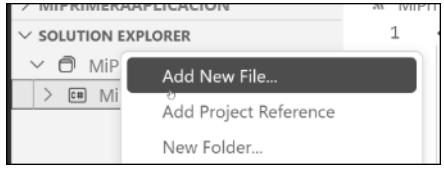
Además de estas normas, es frecuente que los desarrolladores de C# respeten una convención de sintaxis: el uso de PascalCase. Esto indica que el nombre empieza con una mayúscula y cada palabra también empieza con una mayúscula, por ejemplo: `OrdenadorPortatil`. El lenguaje y el compilador no prohíben escribir `ordenadorPortatil`, `Ordenadorportatil` o incluso `ordenadorportatil`, pero estas distintas declaraciones no respetan la convención ampliamente aceptada y aplicada. Finalmente, aunque sea posible, se recomienda evitar caracteres acentuados en el nombre de una clase. Por ejemplo, es mejor llamar a su clase `Peaton` en lugar de `Peatón`, para que el código C# producido sea lo más parecido posible al que tendríamos en inglés.

En el programa de base creado en C# en el capítulo anterior, se ha creado una clase `Program` de manera predeterminada. Podemos constatar que no hay concepto de ámbito ni de `partial` o `static`. Después de declararla, la clase define un bloque donde podemos implementar los datos y los métodos que necesita nuestro programa para funcionar.

1.1.1 Las clases en Visual Studio Code

Para crear una clase en Visual Studio Code, hay que seguir las etapas que aparecen a continuación:

- Expanda la vista **Solution Explorer** del proyecto.
- Colóquese en la carpeta donde desea crear la nueva clase (o directamente en el nombre del proyecto si desea crearla en la raíz).
- Haga clic derecho para seleccionar el elemento de menú **Add a New File**.
- Escriba el nombre de la clase en la pequeña ventana emergente que se abre en la parte superior central de la pantalla (siempre sin espacios ni caracteres especiales).



Añadir una clase nueva con Visual Studio Code

Después de estas operaciones, en la jerarquía situada a la izquierda hay disponible un archivo nuevo que lleva el nombre de la clase seguido por la extensión .cs. De manera predeterminada, este archivo estará abierto.

1.1.2 Herencia

Hay un concepto extremadamente importante en POO: la herencia. En general, si tiene la posibilidad de decir «X es una Y», el equivalente podría ser decir «X hereda de Y». X toma todas las propiedades y comportamientos de Y, pero los particulariza. Vamos a dar un ejemplo concreto: «Un Mac es un ordenador». Entonces, a nivel del desarrollo orientado a objetos, un Mac toma todas las propiedades y comportamientos de un ordenador, pero los particulariza aportando sus propios elementos. Decimos en este caso que `Mac` es una clase hija de la clase `Ordenador`.

En C#, este concepto es central porque todos los elementos que va a manipular obviamente heredan de la clase `System.Object`, que define el comportamiento básico de cualquier objeto. Además, a diferencia de otros lenguajes (como C++), en C# no es posible heredar de varias clases: solo es posible tener una clase madre. Si no se especifica ninguna clase madre, es por definición la clase `System.Object` la que constituye la clase madre (sin que se requiera ninguna operación).

Observación

En C# no se puede heredar de varias clases. Por eso hay que elegir la clase de la que se hereda. Si no se especifica nada, el compilador genera automáticamente, de manera transparente, una herencia de la clase `System.Object`, como se describe con anterioridad. Si especificamos una herencia, eso no quiere decir que la clase herede de `System.Object` y de la clase heredada, sino solo de la clase heredada que sustituye a la herencia generada por el compilador. La clase heredada, en sí misma, hereda de otra clase o directamente de `System.Object`. En última instancia, todas las clases en C# heredan de `System.Object` de una forma u otra.

Para indicar que una clase hereda de otra, hay que usar los dos puntos seguidos de la clase de la que se quiere heredar:

```
class Ordenador { }
class Mac : Ordenador { }
```

Por supuesto, el hecho de que una clase herede de otra no quiere decir que por fuerza tenga acceso a todo lo que se ha definido dentro de la clase madre.

1.1.3 Encapsulación

Todo lo que se encuentra en el interior de una clase se designa mediante un término muy específico: encapsulación. Con ella también aparece el concepto de ámbito, que indica cómo se perciben las cosas desde un punto de vista exterior a la clase.

El ámbito permite definir la visibilidad de un elemento de una clase o de la misma clase. En total hay siete ámbitos en C#:

- `public`: define que el elemento es completamente visible dentro y fuera de la clase.
- `private`: define que el elemento solo es visible en el interior de la clase donde se ha declarado, mientras que es completamente invisible desde el exterior.
- `internal`: define que el elemento solo es visible dentro del proyecto donde se ha declarado. Podemos considerarlo como `public`, pero solo dentro del proyecto donde fue declarado. Otro proyecto que referencie nuestro proyecto no tiene conocimiento de un elemento que se ha declarado como `internal`. De manera predeterminada, si no explica el ámbito explícito en una clase, el compilador selecciona `internal`.
- `protected`: define que el elemento solo es visible en el interior de la clase donde se ha declarado y dentro de su jerarquía de clases hijas. Eso se une con el concepto de la herencia, que veremos más adelante en este capítulo.
- `protected internal`: define una suma entre `protected` e `internal`. Un elemento declarado con este ámbito es visible por la clase interesada, sus clases hijas, así como por todas las otras clases dentro del mismo proyecto. Esto también significa que, si se declara una clase hija fuera del proyecto actual, puede tener acceso a un elemento `protected internal`, al igual que cualquier clase del mismo proyecto.
- `private protected`: define una intersección entre `protected` e `internal`. Un elemento declarado con este ámbito solo es visible por la clase interesada y por sus clases hijas definidas dentro del mismo proyecto. Esto quiere decir que una clase hija definida fuera del proyecto actual no podrá acceder a este elemento.

– `file`: agregado en C# 11, este ámbito define la visibilidad solo dentro del marco del archivo actual. Este ámbito es muy particular porque no está pensado para que lo utilicen directamente los desarrolladores. Existe, principalmente para herramientas de generación automática de código. Sin embargo, en casos muy raros puede resultar útil declarar un elemento que solo existe como parte de un archivo para un algoritmo específico. También cabe señalar que, a diferencia de otros ámbitos, este ámbito solo es válido para la declaración de un tipo. No se puede aplicar a un método, campo o propiedad.

Con todos estos ámbitos, se puede crear la clase que corresponde con precisión a las necesidades de nuestra aplicación, para evitar que ciertos elementos no salgan del perímetro de la clase. Retomando nuestro ejemplo, consideramos que la clase `Ordenador` dispone de un booleano que indica si la máquina está encendida o no. Para evitar que nadie pueda manipular este dato de forma directa, la manera de proceder es definirla como públicamente accesible en modo de lectura, pero privado en lo que respecta a la escritura. En consecuencia, solo un método público, definido en esa clase, como por ejemplo `Encender` o `Apagar`, puede cambiar el valor de este indicador. Así nos protegemos de un cambio de estado no controlado (porque podemos considerar que la operación de extinción necesita efectuar algunas operaciones con antelación antes de transferir el booleano).

1.2 ¿Qué se puede declarar dentro de una clase?

Como ya hemos visto, dentro de una clase podemos declarar dos tipos de elementos: métodos (acciones) y datos. Vamos a ver rápidamente cómo declararlos.

1.2.1 Métodos

Un método traduce una acción que se puede invocar en la clase. Durante la declaración de un método, hay que hacerse las siguientes preguntas:

– ¿Se trata de una acción que debe poder realizarse desde el exterior o solo desde el interior de la clase?

– ¿Se espera un valor de retorno particular?

– ¿Es necesaria alguna información para que este método funcione?

Ya ha tenido una vista previa de una llamada de método en el primer capítulo, en la clase `Consola`: `WriteLine` y `ReadLine`. Estos dos métodos ilustran los puntos antes citados:

– `WriteLine` se debe poder llamar desde el exterior. No esperamos que devuelva un valor después de llamarla, pero es necesario transmitirle la información que queremos escribir.
– `ReadLine` también se debe poder llamar desde el exterior. Necesitamos recuperar solo la información introducida por el usuario, sin que sea preciso transmitirle una información cualquiera.

La sintaxis de declaración de un método dentro de una clase es la siguiente:

```
ÁMBITO [static] TIPO_RETORNO NOMBRE_MÉTODO([PARÁMETROS])
```

El tipo de retorno debe corresponder a un tipo C# conocido. Por ejemplo, si queremos crear un método que realiza la suma de dos números y devuelve el resultado, todo accesible de manera pública, lo declaramos de la siguiente manera:

```
public int Addition(int primero, int segundo) {}
```

Observación

Cuando declaramos un método con un valor de retorno sin escribir el contenido del método, el compilador emite inmediatamente un error de compilación. Esto se debe al hecho de que es obligatorio que cada método que devuelve un resultado contenga una instrucción `return`.

Cuando un método debe devolver un valor, hay que usar la palabra clave `return` para definir el valor que deseamos devolver. La instrucción `return` se puede usar directamente con un valor o podemos utilizar una variable del tipo de retorno esperado. En el caso del ejemplo anterior, son válidas estas dos maneras de escribir el método:

```
public int Suma(int primero, int segundo)
{
    return primero + segundo;
}
public int Suma(int primero, int segundo)
{
```

```
    int resultado = primero + segundo;
    return resultado;
}
```

Un elemento importante que hay que a recordar: del mismo modo que hemos visto en el capítulo anterior con la declaración de clases del mismo nombre dentro del mismo espacio nombres, no se puede declarar dos veces el mismo método dentro de una misma clase. Si los nombres son idénticos y los parámetros también lo son, entonces el compilador C# considera que es el mismo método. El valor de retorno no constituye un elemento distintivo. Así, la declaración de los dos métodos siguientes dentro de la misma clase es imposible y eso provoca un error de compilación:

```
public int Suma (int primero, int segundo)
{
    return primero + segundo;
}
public void Suma (int primero, int segundo)
{
}
```

Observación

Como podemos comprobar en el ejemplo anterior, la palabra clave void especifica que el método no devuelve ningún resultado. El concepto de tipo de retorno es obligatorio y hay que usar esta palabra clave para indicar cuándo no lo hay.

Si el método no toma parámetros, la presencia de paréntesis que se abren y se cierran unidos al nombre del método es, a pesar de todo, necesaria para indicar que se trata de un método:

```
public void MiMetodo()
{
}
```

Dentro de un método que declara su propio bloque, se pueden declarar variables y constantes que se consideran únicamente locales (es decir, visibles dentro del método y de todos sus subbloques, pero invisibles dentro de los bloques padres, directos o indirectos).

Cabe señalar que incluso en el caso de que un método no devuelva ningún valor, es posible utilizar la declaración `return` para detener la ejecución del método. En este caso concreto bastará con escribir la palabra clave `return` y terminar con punto y coma. El código después de la declaración `return` no se ejecutará:

```
public void Suma (int primero, int segundo)
{
    return;
    // la siguiente línea de código nunca se ejecutará
    int resultado = primero + segundo;
}
```

1.2.2 Declarar un dato

Hay dos maneras de declarar un dato dentro de una clase: mediante una propiedad o mediante un campo. Las dos son completamente legítimas, pero no responden a las mismas necesidades.

El caso más sencillo es la declaración de un campo. Un campo de una clase se define de la siguiente manera:

```
ÁMBITO TIPO NOMBRE_DEL_MIEMBRO;
```

Observación

El concepto de ámbito no es obligatorio y es posible omitirlo. En ausencia de la definición de ámbito, es `private` el utilizado por un campo en el compilador. Sin embargo, es muy recomendable añadirla por motivos de claridad.

Por ejemplo, si en nuestra clase `Ordenador` queremos almacenar el año de compra bajo la forma de un entero accesible para todos, podemos crear un campo como este:

```
public int anoCompra;
```

Como podemos ver aquí arriba, la convención de sintaxis recomendada para la escritura de los campos es *lower camel casing*, precedido de un guion bajo. Esta convención indica que la primera letra es minúscula, pero todas las palabras siguientes empiezan por su propia mayúscula.

Capítulo 3
Novedades de ASP.NET Core

1. Introducción

ASP.NET existe desde 2002 y se han introducido muchos cambios en el framework desde su primera versión. Es importante recordar una cosa sobre ASP.NET Core: la nueva plataforma web de Microsoft no es en absoluto una continuación de la versión 4.6 del framework que ya conocemos, sino más bien una renovación que debería marcar una nueva era para la tecnología de Microsoft en la Web moderna.

Algunos dirán que el framework no ha cambiado tanto (especialmente la parte MVC), pero es realmente 'bajo el capó' donde los cambios han sido más profundos, empezando por el espacio de nombres `System.Web`, que ya no existe. A continuación, anunciado como multiplataforma, ASP.NET Core es más modular que en años anteriores. A través de ***NuGet*** para los componentes del servidor y después a través de ***Grunt*** o ***Gulp*** para la parte cliente del sitio web, el nuevo framework también se beneficia de un nuevo runtime, llamado ***CoreCLR***, que permite ejecutar una aplicación web de Microsoft en Linux o Mac.

2. Nuevas herramientas de código abierto

ASP.NET Core viene con toda una nueva gama de herramientas de código abierto para gestionar nuevos proyectos web. Afortunadamente para los desarrolladores, todas estas herramientas se han reunido en una única interfaz de línea de comandos: dotnet.

2.1 El entorno de ejecución dotnet

dotnet ha sido diseñado para que las aplicaciones .NET funcionen en plataformas Windows, Mac y Linux, sin tener que desarrollar un runtime distinto para cada una de ellas. Es tanto un entorno de ejecución como un SDK, con todo lo necesario para hacer funcionar aplicaciones web ASP.NET multiplataforma.

Totalmente orientado ***package-first***, Microsoft ha llevado el concepto de modularidad un paso más allá, permitiendo incluso que el entorno de ejecución incorpore, gestione y cree automáticamente los paquetes que necesita a través de NuGet. dotnet se puede usar en diferentes frameworks (.NET Core o el framework .NET Full) y generar paquetes NuGet directamente. Además, dotnet incluye el nuevo motor de ejecución CoreCLR, diseñado específicamente para los problemas de compatibilidad en otras plataformas.

dotnet está integrado en Visual Studio 2015 para ofrecer una experiencia más rica al desarrollador, pero el entorno de ejecución también se puede controlar desde la línea de comandos. En un proyecto ASP.NET Core, puede añadir herramientas de Microsoft y utilizar dotnet para controlar el proyecto desde la línea de comandos. Estas herramientas se añaden al mismo tiempo que el paquete NuGet en el archivo *.csproj:*

```
<ItemGroup>
    <DotNetCliToolReference Inclu-de="Microsoft.VisualStudio.Web.
CodeGeneration.Tools" Version="2.0.4" />
    <DotNetCliToolReference Include="BundlerMinifier.Core"
Version="2.0.238" />
  </ItemGroup>
```

Los comandos declarados se pueden utilizar, por ejemplo, para lanzar una herramienta de generación de código, antes conocida como *scaffolding*. Esto se puede utilizar para generar el controlador y las vistas que corresponden a un modelo de datos muy específico. El comando se puede utilizar a través de `dotnet <command>`:

```
dotnet Microsoft.VisualStudio.Web.CodeGeneration.Tools
```

Dentro de la propia aplicación, es posible utilizar servicios de dotnet a través de ciertas interfaces C# disponibles mediante inyección de dependencias. Servicios como `IServiceEnvironment` se pueden utilizar para modificar la configuración del entorno de ejecución mientras se ejecuta la propia aplicación.

La utilidad `dotnet` también contiene una lista de comandos predefinidos para realizar determinadas acciones en el servidor ASP.NET Core:

– `dotnet` new: inicializa un proyecto sencillo de consola C#.
– `dotnet restore`: restaura las dependencias del proyecto según *.csproj*.
– `dotnet build`: crea la aplicación .NET Core.
– `dotnet publish`: publica una aplicación .NET Core.
– `dotnet run`: lanza la aplicación desde el código fuente.
– `dotnet test`: ejecuta las pruebas utilizando el *test runner* definido en el *.csproj*.
– `dotnet pack`: crea un paquete NuGet a partir del código fuente.

Esta herramienta admite varios procesos para lanzar la aplicación ASP.NET Core. El primero consiste simplemente en restaurar los paquetes necesarios para la aplicación y, a continuación, lanzar el proyecto desde el código fuente.

```
dotnet new
dotnet restore
dotnet run
```

Sin embargo, la utilidad también se puede utilizar para lanzar la aplicación directamente desde una DLL, una vez generado el proyecto.

```
dotnet build
dotnet run bin/Debug/netcoreapp1.0/test-app.dll
```

Con los dos procesos anteriores, la herramienta demuestra que es capaz de ejecutar varios tipos de aplicaciones:

– Aplicaciones "portátiles", es decir, aplicaciones que dependen de la versión de .NET Core instalada en la máquina. Esto significa que este tipo de aplicaciones se podrán ejecutar en distintas instalaciones de .NET Core, independientemente de los sistemas operativos utilizados. Las aplicaciones "portátiles" también permiten centralizar las librerías .NET: por tanto, solo incrustarán librerías externas.
– Aplicaciones "autónomas", es decir, aplicaciones que contienen todas las dependencias que necesitan para funcionar, incluido el runtime de .NET Core, que es parte integrante de la aplicación. Esto facilita el despliegue de la aplicación, pero hace que el paquete sea más pesado. Además, la aplicación debe especificar la versión del runtime que se utilizará en el *.csproj*.

2.2 La utilidad dotnet restore

Un proyecto web ASP.NET Core tiene varias dependencias de paquetes externos, a menudo de NuGet. Para poner en marcha una aplicación web, primero hay que restaurar los paquetes necesarios para que el proyecto funcione correctamente. Esta es una de las novedades de ASP.NET Core: el proyecto solo incorpora lo que necesita, por lo que ha sido necesario diseñar una herramienta que permita restaurar estas dependencias, de forma multiplataforma y totalmente transparente. dotnet restore permite llevar a cabo esta restauración.

Integrado en .NET Core e instalado con dotnet, dotnet restore permite analizar el proyecto ASP.NET Core y recuperar de la nube los paquetes que necesita. Visual Studio 2015 utiliza automáticamente esta utilidad cuando se actualizan las dependencias del proyecto. Sin embargo, puede iniciar el proceso manualmente mediante el siguiente comando:

```
> dotnet restore
```

Observación

Hay que tener en cuenta, sin embargo, que en la consola desde la que se lanza el comando es necesario estar ubicado en la raíz del proyecto. dotnet restore inspeccionará el archivo .csproj para identificar las dependencias a restaurar.

Gracias a dotnet restore y a su facilidad de uso, es muy sencillo restaurar las dependencias de un proyecto ASP.NET Core, independientemente de la plataforma.

2.3 Gestión de paquetes NuGet con dotnet pack

La herramienta `dotnet` pack se utiliza para generar un paquete NuGet a partir de un proyecto .NET. Se puede utilizar de varias maneras para gestionar los paquetes NuGet. En primer lugar, podemos utilizar el siguiente comando, en la raíz de un proyecto .NET, para simplemente generar un paquete NuGet:

```
dotnet pack mi_proyecto.csproj
```

Este comando generará un paquete NuGet a partir del proyecto especificado y lo guardará en el directorio `bin\Debug` o `bin\Release`, dependiendo del modo de compilación utilizado.

Para incluir información sobre la versión y los metadatos en el paquete NuGet generado, podemos utilizar las opciones --`version` y --`metadata`:

```
dotnet pack mi_proyecto.csproj --version 1.2.3
--metadata authors="John Doe"
```

En este ejemplo, hemos especificado la versión `1.2.3` del paquete y añadido un metadato `authors` con el valor "`John Doe`".

Para publicar el paquete NuGet generado en un repositorio NuGet, podemos utilizar la opción --`publish`, especificando la URL del repositorio:

```
dotnet pack mi_proyecto.csproj --publish https://mynugetrepo.com
```

Este comando publicará el paquete NuGet generado en el repositorio NuGet especificado.

Para más información, puede consultar la documentación oficial o utilizar el comando `dotnet pack --help` para mostrar la lista completa de opciones disponibles:

– `--output`: directorio de salida de los paquetes generados.
– `--no-build`: genera un paquete NuGet sin lanzar la generación del proyecto .NET.
– `--no-restore`: genera un paquete NuGet sin restaurar los paquetes NuGet del proyecto.
– `--include-symbols`: incluye símbolos de compilación junto al paquete generado en la carpeta de salida.
– `--include-source`: incluye archivos PDB y archivos fuente. Las fuentes irán a la carpeta `src`.
– `--serviceable`: define el nivel de mantenimiento del paquete.
– `--nologo`: no muestra el logotipo cuando se inicia el comando.
– `--interactive`: permite esperar la interacción del usuario si es necesario.
– `--verbosity`: indica la verbosidad de los registros que se muestran al ejecutar el comando.
– `--version-suffix`: define el valor de la propiedad `$(VersionSuffix)`que se utilizará al generar el proyecto.
– `--configuration`: define la configuración que se utilizará durante la generación. Los valores pueden ser `Debug` o `Release`.
– `--use-current-runtime`: define si se debe utilizar el runtime actual como runtime de destino.

El comando `dotnet` pack es muy útil para gestionar paquetes NuGet para proyectos .NET y permite a los desarrolladores crear y publicar paquetes fácilmente.

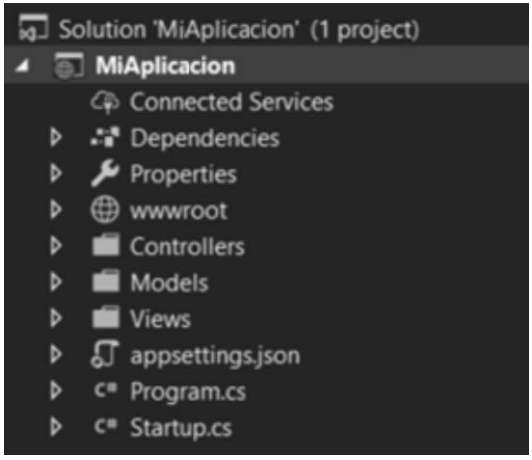
3. La estructura de una solución

Una solución ASP.NET Core es la base de un proyecto web que utiliza tecnologías Microsoft. Permite desplegar rápidamente un sitio y estructurar el código utilizado para ejecutar la aplicación. Una solución puede incluir código tanto del lado del servidor como del lado del cliente, a la vez que incluye mecanismos para separar ambas partes. Este capítulo examinará los diferentes componentes de una solución ASP.NET Core y explicará sus funciones en la configuración o despliegue de la aplicación web.

3.1 Archivos .csproj

Un proyecto ASP.NET Core implementa una nueva filosofía de diseño de aplicaciones web de Microsoft que se inspira en gran medida en el código abierto.

La nueva plantilla tiene este aspecto:



Nueva plantilla de proyecto ASP.NET Core

Lo primero que hay que saber es que la disposición de los proyectos en la solución no es fija. De hecho, mediante un sistema de referencia, es muy fácil modificar la ubicación de los proyectos para seguir sus propias convenciones de nomenclatura y estructura. Un proyecto básico ASP.NET Core MVC se compone de los siguientes elementos:

– **Una solución**: es el vínculo entre todos sus proyectos. El archivo .sln permite agrupar en un único archivo toda la información importante sobre sus proyectos: nombre y ruta del proyecto, configuración de compilación del proyecto (Debug, Release, etc.). También reúne diversa información contextual sobre la solución, como la versión mínima de Visual Studio necesaria para abrir el proyecto. El formato del archivo es muy similar a YAML.

```
Microsoft Visual Studio Solution File, Format Version 12.00
# Visual Studio 15
VisualStudioVersion = 15.0.28010.2016
MinimumVisualStudioVersion = 10.0.40219.1
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
"MiAplicacion", "MiAplicacion\MiAplicacion.csproj",
"{GUID-PROJET}
EndProject
Global
   GlobalSection(SolutionConfigurationPlatforms) = preSolution
       Debug|Any CPU = Debug|Any CPU
       Release|Any CPU = Release|Any CPU
   EndGlobalSection
   GlobalSection(ProjectConfigurationPlatforms) = postSolution
       {GUID-PROJET}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
          {GUID-PROJET}.Debug|Any CPU.Build.0 = Debug|Any CPU
       {GUID-PROJET}.Release|Any CPU.ActiveCfg = Release|Any CPU
       {GUID-PROJET}.Release|Any CPU.Build.0 = Release|Any CPU
   EndGlobalSection
   GlobalSection(SolutionProperties) = preSolution
       HideSolutionNode = FALSE
   EndGlobalSection
   GlobalSection(ExtensibilityGlobals) = postSolution
       SolutionGuid = {1D73F40D-E395-4018-B0A3-DD55BE367A54}
   EndGlobalSection
EndGlobal
```

- La sección **Connected Services**: se utiliza para conectar servicios externos al proyecto, como Application Insights o Azure Key Vault.
- La sección **Dependencies**: agrupa todas las dependencias del proyecto y las clasifica en tres categorías: analizadores (analizadores de código, por ejemplo), paquetes NuGet y dependencias del SDK .NET Core.
- La sección **Properties**: incluye un archivo de configuración, **launchSettings.json**, utilizado para configurar los perfiles de inicio del proyecto. Este archivo se describe con más detalle más adelante en esta sección.
- La carpeta **wwwroot**: esta carpeta contiene todos los archivos destinados únicamente al cliente web: archivos CSS, JavaScript, imágenes y recursos de todo tipo.
- Carpetas **Controllers**, **Models** y **Views**: carpetas que contienen el código del proyecto web MVC. La asignación de nombres es muy importante, porque permite al framework encontrar los controladores según las plantillas de enrutamiento definidas en la configuración del proyecto.
- Archivos **appsettings.json**: archivos de configuración del proyecto para diferentes entornos. Estos archivos contendrán, por ejemplo, cadenas de conexión a bases de datos o constantes que se pueden configurar fácilmente fuera del código fuente o que pueden diferir en función del entorno de ejecución de la aplicación.
- La clase **Program.cs**: el punto de entrada de la aplicación. Aquí es donde se lanza el servidor web Kestrel con la configuración inicial de la infraestructura necesaria para que funcione correctamente (puerto, integración con IIS, etc.).
- La clase **Startup.cs**: clase de configuración del proyecto web. Define los servicios que se utilizarán en toda la aplicación. También define el pipeline HTTP para las peticiones entrantes y salientes.