Chrome
Mozilla FireFox
I Explorer
Safari
Opera

Historia de los navegadores

El primer navegador fue desarrollado por Tim Berners-Lee, en la CERN, en 1990; el navegador web llamado ***WorldWideWeb*** era bastante sofisticado y gráfico, pero solo funcionaba en estaciones NexT.

El navegador Mosaic, que funcionaba inicialmente en entornos Unix sobre XFree86 (X11), fue el primero que se extendió debido a que pronto el National Center for Supercomputing Applications (NCSA) preparó versiones para Windows y Macintosh.

Netscape Navigator al poco tiempo entró en el mercado y rápidamente superó en capacidades y velocidad al Mosaic. Este navegador tuvo la ventaja de funcionar en casi todos los sistemas Unix, y también en entornos Windows.

Vs

Internet Explorer (anteriormente Spyglass Mosaic) fue la apuesta tardía de Microsoft para entrar en el mercado y consiguió desbancar al Netscape Navigator entre los usuarios de Windows, debido a la integración del navegador con el sistema operativo, llegando a poseer cerca del 95% de la cuota de mercado. Netscape Communications Corporation liberó el código fuente de su navegador, naciendo así el proyecto Mozilla.

Mozilla (Mozilla Application Suite) fue reescrito desde cero tras decidirse a desarrollar y usar como base un nuevo conjunto de widgets multiplataforma basado en Extensible Markup Language (XML) llamado XUL y esto hizo que tardara bastante más en aparecer de lo previsto inicialmente, apareciendo una versión 1.0 de gran calidad y para muchísimas plataformas a la vez el 5 de junio de 2002.

El 7 de enero de 2003, Apple lanzó al mercado el navegador web Safari. Este navegador se hace con casi la totalidad del mercado de las microcomputadoras Mac, debido a su velocidad y gran cantidad de actualizaciones.

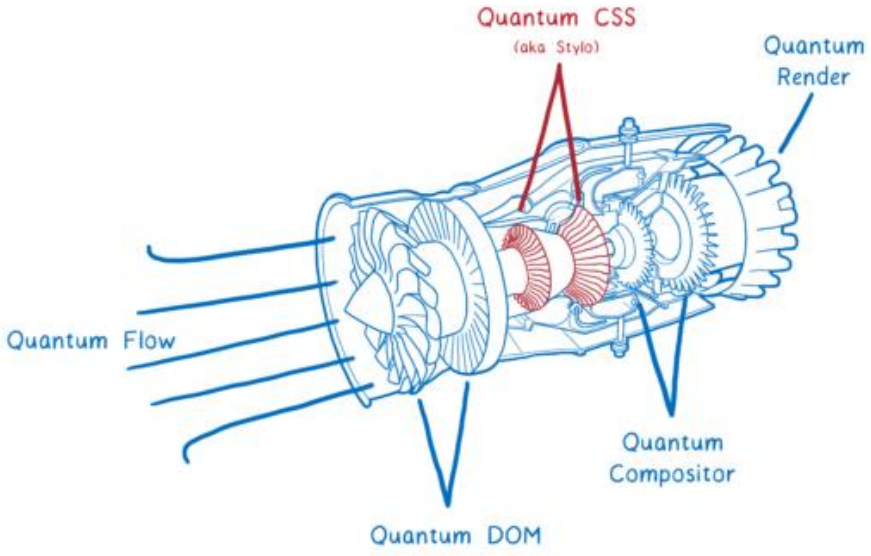
El 2 de septiembre de 2008, Google Chrome aparece en el mercado. Es el navegador web desarrollado por Google y compilado con base en componentes de código abierto como el motor de renderizado de ***WebKit*** y su estructura de desarrollo de aplicaciones (framework). El nombre del navegador deriva del término usado para el marco de la interfaz gráfica de usuario ("chrome"). En diciembre de 2011, Chrome superó a Internet Explorer 8.0 como el navegador más utilizado a nivel mundial.

El 29 de julio de 2015, Microsoft lanza Microsoft Edge como sucesor de Internet Explorer. Es una versión mejorada, modernizada y distinta de Internet Explorer con una línea de desarrollo independiente. En 2020, Microsoft reconstruyó a Microsoft Edge, ahora bajo Chromium, la cual ha tenido mayor aceptación a los usuarios.

INFORMACIÓN ADICIONAL

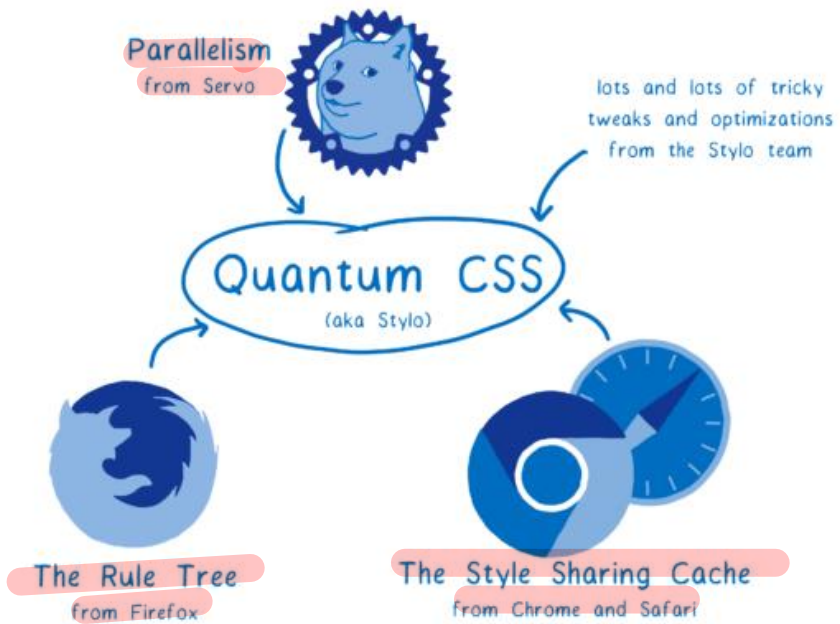
Es posible que hayas oído hablar de ***Project Quantum*** … es una reescritura importante de las partes internas de Firefox para hacer que Firefox sea rápido. Estamos intercambiando partes de nuestro buscador experimental, Servo, y realizando mejoras masivas en otras partes del motor.

El proyecto ha sido comparado con el reemplazo de un motor a reacción mientras el avión aún está en vuelo. Estamos implementando los cambios, componente por componente, para que pueda ver los efectos en Firefox tan pronto como cada componente esté listo.



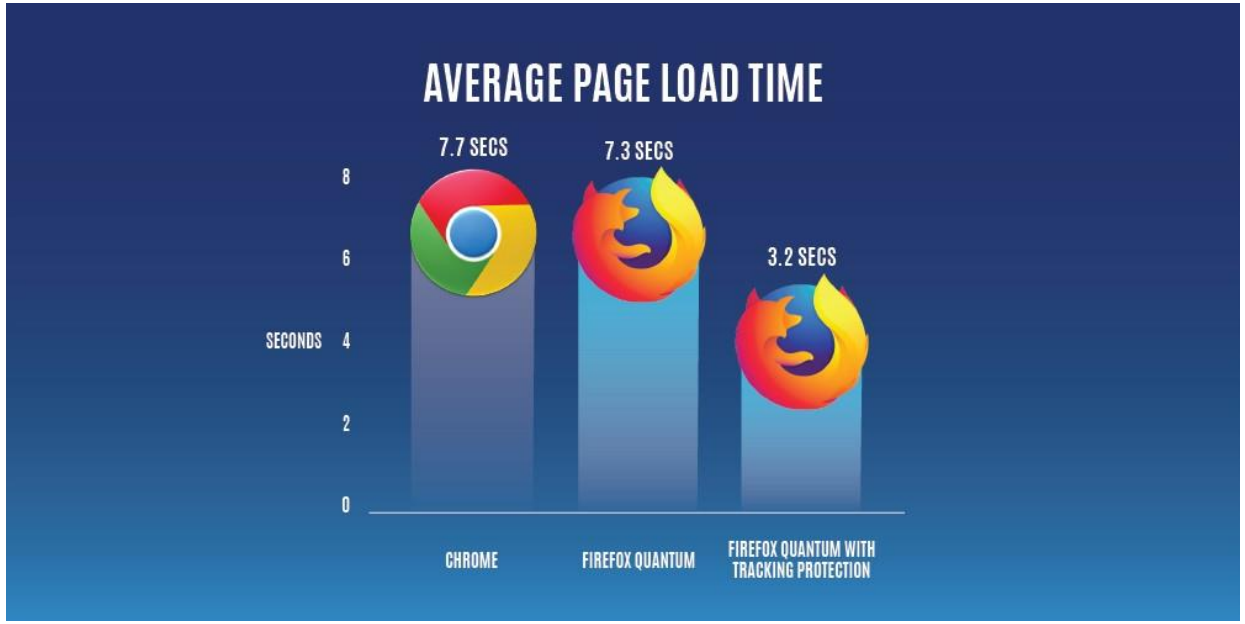
El primer componente principal de Servo, un nuevo motor de CSS llamado Quantum CSS (anteriormente conocido como Stylo), ahora está disponible para probar en nuestra versión Nightly.
Puede asegurarse de que esté activada yendo ***about:configay*** estableciendo ***layout.css.servo.enableden*** verdadero.

Este nuevo motor reúne innovaciones de vanguardia de cuatro navegadores diferentes para crear un nuevo motor supercss.



Principales características

- ***Trabaja en paralelo***: el nuevo motor de renderizado, llamado Servo, aprovecha todos los núcleos de proceso de tu CPU, así que trabaja en paralelo.
- ***Cada pestaña, un proceso (más o menos)***: el proyecto Electrolysis es otro de los componentes clave del nuevo Firefox Quantum, considera cada pestaña como un proceso separado
- ***Nuevo look***: también hay cambios en el aspecto visual del navegador, que tiene menús más simples y mejor organizados
- ***Extensiones***: otro de los cambios cruciales en la nueva versión es hacia las llamadas WebExtensions, un nuevo tipo de extensiones o complementos que hacen que Firefox pueda gestionarlas
- ***Privacidad, velocidad y consumo de datos***: como explican los repsonsables de Mozilla, la tecnología Tracking Protection de Firefox bloquea las peticiones de los dominios que monitorizan nuestra actividad en la web



Componentes principales del navegador

- **Interfaz de usuario:** incluye la barra de direcciones, el botón de avance/retroceso, el menú de marcadores, etc.
- **Motor de búsqueda:** coordina las acciones entre la interfaz y el motor de renderización.
- **Motor de renderización:** es responsable de mostrar el contenido solicitado. Por ejemplo, si el contenido solicitado es HTML, será el responsable de analizar el código HTML y CSS
- **Red:** es responsable de las llamadas de red, como las solicitudes HTTP.
- **Servidor de la interfaz:** permite presentar widgets básicos, como ventanas y cuadros combinados. Muestra una interfaz genérica que no es específica de ninguna plataforma.
- **Intérprete de JavaScript:** permite analizar y ejecutar el código JavaScript.
- **Almacenamiento de datos:** es una capa de persistencia. El navegador necesita guardar todo tipo de datos en el disco duro (por ejemplo, las cookies). La nueva especificación de HTML (HTML5) define el concepto de "base de datos web", que consiste en una completa (aunque ligera) base de datos del navegador.

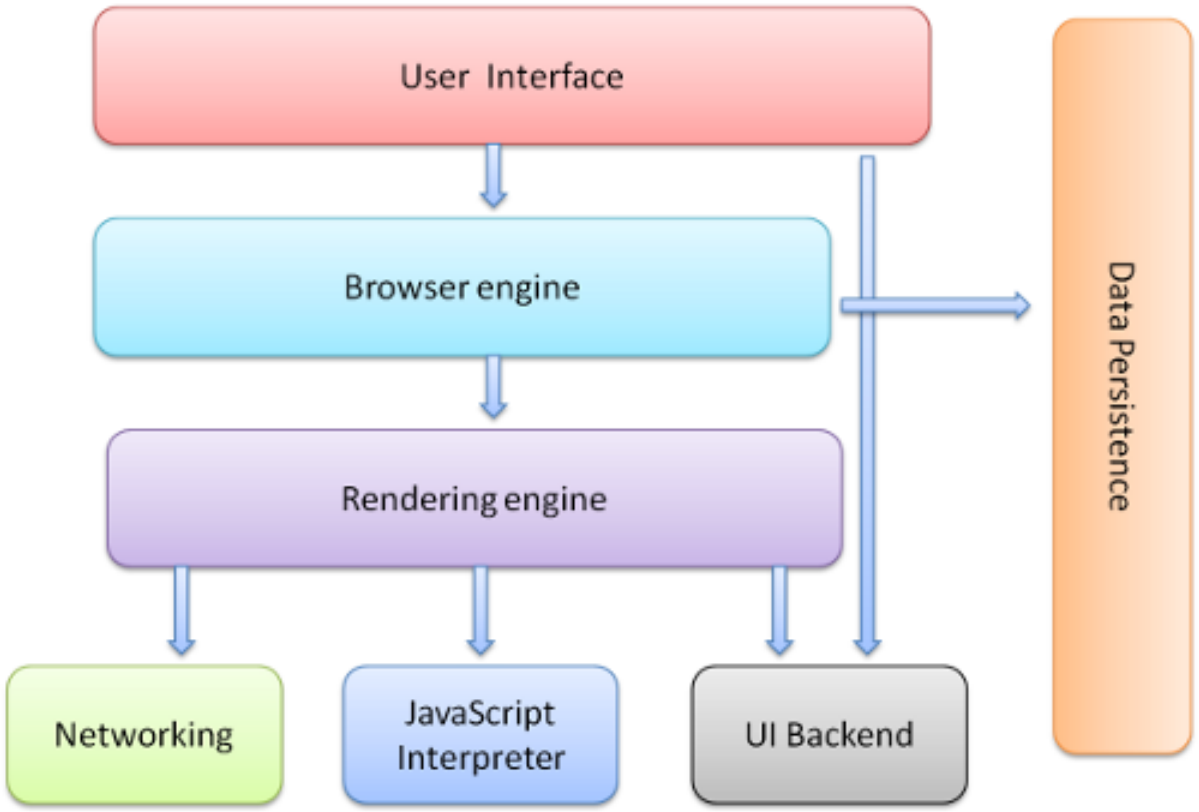


Figura : componentes principales del navegador

Motores de navegadores web: Gecko, Trident, WebKit y otros

Un motor de navegador web (motor de renderizado) o del inglés web browser engine es software que toma contenido marcado (como HTML, XML, archivos de imágenes, etc.) e información de formateo (como CSS, XSL, etc.) y luego muestra el contenido ya formateado en la pantalla. El motor «pinta» en el área de contenido de una ventana, la cual es mostrada en un monitor.

Por otro lado, están los interpretes de javascript o motor de Javascript, que, aunque cada motor de navegador suele tener su propio interprete javascript no son realmente lo mismo. Mientras que la función del motor del navegador es ***renderizar y pintar la parte gráfica de la web***, el motor de javascript es el interprete del código y quien ejecuta un script en función de unas instrucciones.

Cada navegador tiene un motor de renderizado. En Chrome, se llama ***Blink***. En Edge, se llama ***EdgeHTML***. En Safari, se llama ***WebKit***. Y en Firefox, se llama ***Gecko***.

Proceso de renderizado

El motor de renderización empieza a analizar el documento HTML y convierte las etiquetas en nodos DOM en un árbol denominado ***"árbol de contenido"***. Analiza los datos de estilo, tanto en los archivos CSS externos como en los elementos de estilo. Los datos de estilo, junto con las instrucciones visuales del código HTML, se utilizan para crear otro árbol: el **"árbol de renderización"**.

Ejemplos del flujo principal

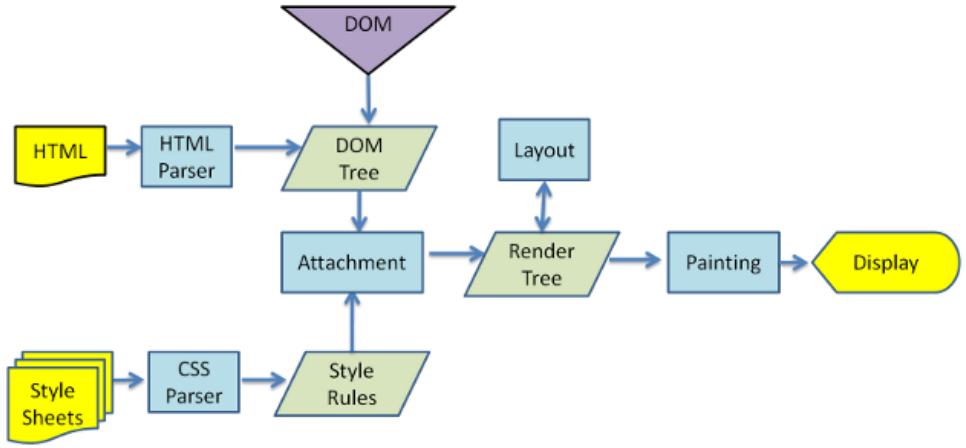
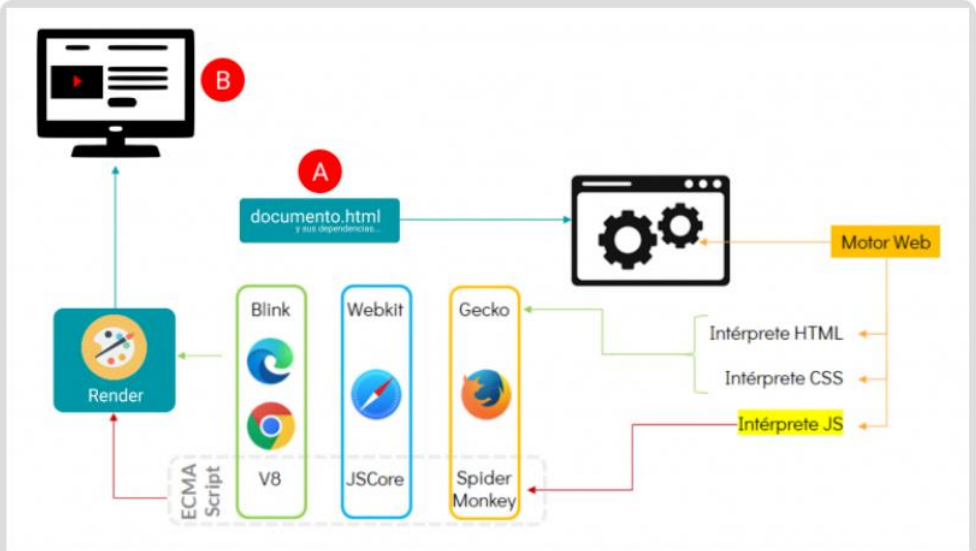


Figura : flujo principal de WebKit



Arquitectura del proceso de carga de un sitio o WebApp, en un navegador web.

Analizar un documento significa traducirlo a una estructura que tenga sentido, es decir, algo que el código pueda comprender y utilizar. El resultado del análisis suele ser un árbol de nodos que representa la estructura del documento. Este árbol se denomina "árbol de análisis" o "árbol de sintaxis".

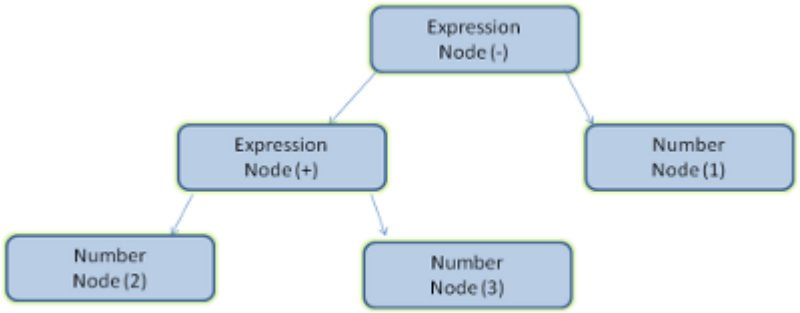


Figura : nodo de árbol de expresión matemática

Gramáticas

El análisis se basa en las reglas de sintaxis por las que se rige el documento, es decir, el lenguaje o el formato en el que está escrito.

Analizador: combinación de analizadores léxicos

El proceso de análisis se puede dividir en dos subprocesos: análisis léxico y análisis sintáctico.

El análisis léxico es el proceso de descomponer los datos de entrada en tokens.

El análisis sintáctico es la aplicación de las reglas sintácticas del lenguaje.

Traducción

Muchas veces, el árbol de análisis no es el producto final. El análisis se utiliza frecuentemente en la traducción, es decir, en la conversión del documento de entrada a otro formato. Un ejemplo sería la compilación.

Definición de gramática HTML

- Es la sintaxis y el vocabulario definidos en las especificaciones creadas por la organización W3C. La versión actual es HTML5.

- el lenguaje HTML es bastante similar al XML. Hay una gran variedad de analizadores de XML disponibles. Existe una variación XML del lenguaje HTML, el XHTM

- La diferencia radica en que el lenguaje HTML es más "permisivo", ya que permite omitir ciertas etiquetas que se añaden de forma implícita, a veces se omite el principio o el final de las etiquetas, etc.

<HTML>
</HTML>

Capítulo 1. Introducción

1.1. ¿Qué es HTML?

- Definiéndolo de forma sencilla, *"HTML es lo que se utiliza para crear todas las páginas web de Internet"*. Más concretamente, HTML es el *lenguaje* con el que se *"escriben"* la mayoría de las páginas web.

- El lenguaje HTML es un estándar reconocido en todo el mundo y cuyas normas define un organismo sin ánimo de lucro llamado World Wide Web Consortium, más conocido como **W3C**. Como se trata de un estándar reconocido por todas las empresas relacionadas con el mundo de Internet, una misma página HTML se visualiza de forma muy similar en cualquier navegador de cualquier sistema operativo.

- El propio **W3C** define el lenguaje HTML como *"un lenguaje reconocido universalmente y que permite publicar información de forma global"*.

1.2. Breve historia de HTML

- El origen de HTML se remonta a 1980, cuando el físico **Tim Berners-Lee**, trabajador del CERN (*Organización Europea para la Investigación Nuclear*) propuso un nuevo sistema de *"hipertexto"* para compartir documentos.

- El primer documento formal con la descripción de HTML se publicó en 1991 bajo el nombre *HTML Tags* (*Etiquetas HTML*) y todavía hoy puede ser consultado online a modo de *reliquia informática*.

- A partir de 1996, los estándares de HTML los publica un organismo de estandarización llamado W3C (*World Wide Web Consortium*). Esta revisión incorpora los últimos avances como *applets* de Java y texto que fluye alrededor de las imágenes.

- En el año 2004, las empresas Apple, Mozilla decidieron organizarse en una nueva asociación llamada WHATWG (*Web Hypertext Application Technology Working Group*). La cual se centra en el futuro estándar HTML 5, cuyo primer borrador oficial se publicó el 22 de enero de 2008.

- La W3C ha continuado con la estandarización de XHTML, una versión *avanzada* de ***HTML*** y basada en ***XML***. La primera versión de XHTML se denomina XHTML 1.0 y se publicó el 26 de Enero de 2000.

1.3. HTML y XHTML

El lenguaje XHTML es muy similar al lenguaje HTML. De hecho, XHTML no es más que una adaptación de HTML al lenguaje XML. Técnicamente, HTML es descendiente directo del lenguaje SGML, mientras que XHTML lo es del XML (que, a su vez, también es descendiente de SGML).

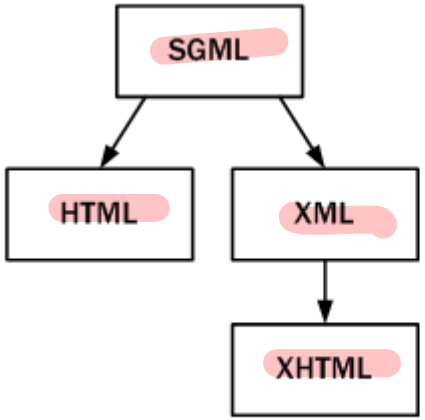


Figura 1.1 Esquema de la evolución de HTML y XHTML

2.1. Lenguajes de etiquetas

HTML es un **lenguaje de etiquetas** (también llamado **lenguaje de marcado**) y las páginas web habituales están formadas por cientos o miles de pares de etiquetas. De hecho, las letras "ML" de la sigla HTML significan *"markup language"*, que es como se denominan en inglés a los *lenguajes de marcado*. Además de HTML, existen muchos otros lenguajes de etiquetas como XML, SGML, etc.

Aunque existen algunas excepciones, en general las etiquetas se indican por pares y se forman de la siguiente manera:

- Etiqueta de apertura: carácter <, seguido del nombre de la etiqueta (sin espacios en blanco) y terminado con el carácter >
- Etiqueta de cierre: carácter <, seguido del carácter /, seguido del nombre de la etiqueta (sin espacios en blanco) y terminado con el carácter >

```
<nombre_etiqueta> ... </nombre_etiqueta>
```

INFORMACIÓN ADICIONAL

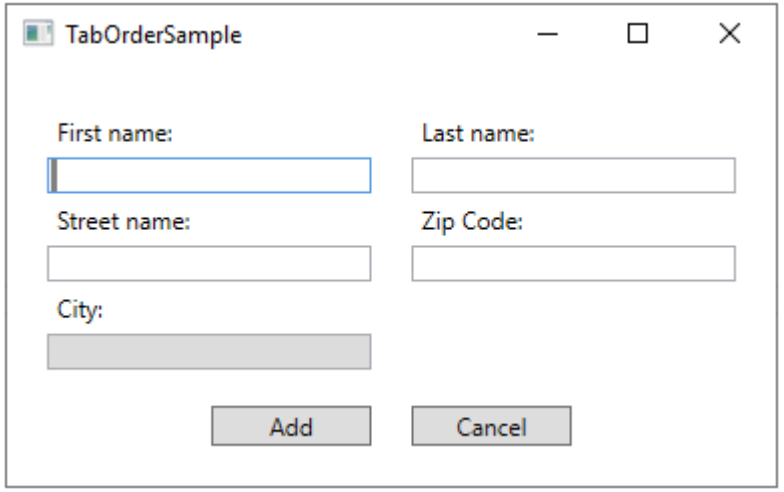
XML es un lenguaje de marcado similar a HTML. Significa Extensible Markup Language (Lenguaje de Marcado Extensible) y es una especificación de W3C como lenguaje de marcado de propósito general. Esto significa que, a diferencia de otros lenguajes de marcado, XML no está predefinido, por lo que debes definir tus propias etiquetas. El propósito principal del lenguaje es compartir datos a través de diferentes sistemas, como Internet.

Ejemplos de uso:

- Creación de aplicaciones de escritorio mediante el uso de la biblioteca ***WPF*** (***Windows Presentation Fundation***) para el desarrollo de interfaces gráficas definidas en su mayoría en el espacio de nombres System.Windows, mediante el uso de **XAML** (**Extensible Aplication Markup Language** o XML para aplicaciones).
- Programación Android mediante Visual Studio
- Programación Android e IOS mediante Xamarin

Windows Presentation Fundation

```xml
<Window x:Class="WpfTutorialSamples.Control_concepts.TabOrderSample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfTutorialSamples.Control_concepts"
    mc:Ignorable="d"
    Title="TabOrderSample" Height="250" Width="400">
    <Grid Margin="20">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="20" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <StackPanel>
        <Label>First name:</Label>
        <TextBox TabIndex="0" />
        <Label>Street name:</Label>
        <TextBox TabIndex="2" />
        <Label>City:</Label>
        <TextBox TabIndex="5" IsReadOnly="True" IsTabStop="False" Background="Gainsboro" />
    </StackPanel>
    <StackPanel Grid.Column="2">
        <Label>Last name:</Label>
        <TextBox TabIndex="1" />
        <Label>Zip Code:</Label>
        <TextBox TabIndex="4" />
    </StackPanel>
    <Button Grid.Row="1" HorizontalAlignment="Right" Width="80">Add</Button>
    <Button Grid.Row="1" Grid.Column="2" HorizontalAlignment="Left" Width="80">Cancel</Button>
    </Grid>
</Window>
```



Visual Studio

activity_splash_screen.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.academiaandroid.listacontactos.SplashScreenActivity">

    <ImageView
        android:layout_width="250dp"
        android:layout_height="150dp"
        android:id="@+id/imageView"
        android:src="@drawable/g4569"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true" />
</RelativeLayout>
```

Design | Text

Preview

Nexus 4 | AppTheme | SplashScreen | 21

Xamarin

Here's the XAML file for the program whose screenshots you've just seen:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PlatformVisuals.PlatformVisualsPage"
             Title="Visuals">

    <StackLayout Padding="10,0">
        <Label Text="Hello, Xamarin.Forms!"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <Button Text = "Click Me!"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center" />

        <Switch VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center" />

        <Slider VerticalOptions="CenterAndExpand" />
    </StackLayout>

    <ContentPage.ToolbarItems>
        <ToolbarItem Text="edit" Order="Primary">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                            iOS="edit.png"
                            Android="ic_action_edit.png"
                            WinPhone="Images/edit.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>

        <ToolbarItem Text="search" Order="Primary">
            <ToolbarItem.Icon>
```

2.2. El primer documento HTML

Las páginas HTML se dividen en dos partes: la cabecera y el cuerpo. La cabecera incluye información sobre la propia página, como por ejemplo su título y su idioma. El cuerpo de la página incluye todos sus contenidos, como párrafos de texto e imágenes.

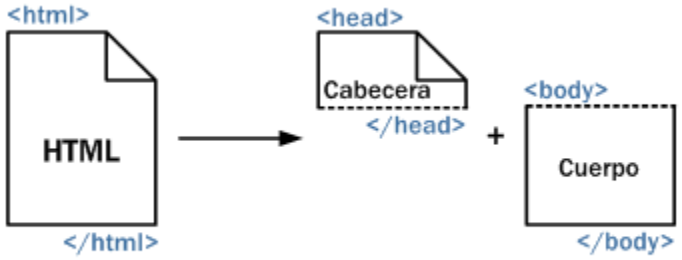


Figura 2.4 Esquema de las etiquetas principales que contiene un documento HTML

- HTML define 91 etiquetas que los diseñadores pueden utilizar para marcar los diferentes elementos que componen una página:

a, abbr, acronym, address, applet, area, b, base, basefont, bdo, big, blockquote, body, br, button, caption, center, cite, code, col, colgroup, dd, del, dfn, dir, div, dl, dt, em, fieldset, font, form, frame, frameset, h1, h2, h3, h4, h5, h6, head, hr, html, i, iframe, img, input, ins, isindex, kbd, label, legend, li, link, map, menu, meta, noframes, noscript, object, ol, optgroup, option, p, param, pre, q, s, samp, script, select, small, span, strike, strong, style, sub, sup, table, tbody, td, textarea, tfoot, th, thead, title, tr, tt, u, ul, var.

- De todas las etiquetas disponibles, las siguientes se consideran obsoletas y no se pueden o deben utilizar:

applet, basefont, center, dir, font, isindex, menu, s, strike, u.

La etiqueta ***<a>*** por ejemplo se emplea para incluir un enlace en una página. Utilizando sólo la etiqueta ***<a>*** no es posible establecer la dirección a la que apunta cada enlace. Como no es viable crear una etiqueta por cada enlace diferente, la solución consiste en personalizar las etiquetas HTML mediante cierta información adicional llamada ***atributos***.

Los atributos comunes se dividen en cuatro grupos según su funcionalidad:

- ***Atributos básicos***: se pueden utilizar prácticamente en todas las etiquetas HTML.
- ***Atributos de eventos***: sólo se utilizan en las páginas web dinámicas creadas con JavaScript.
- ***Atributos de foco***: relacionados principalmente con la accesibilidad de los sitios web.
- ***Atributos para internacionalización***: los utilizan las páginas que muestran sus contenidos en varios idiomas.

2.3.1. Atributos básicos

Atributo	Descripción
id = "texto"	Establece un identificador único a cada elemento dentro de una página HTML
class = "texto"	Establece la clase CSS que se aplica a los estilos del elemento
style = "texto"	Establece de forma directa los estilos CSS de un elemento
title = "texto"	Establece el título a un elemento (mejora la accesibilidad y los navegadores lo muestran cuando el usuario pasa el ratón por encima del elemento)

2.3.2. Atributos de eventos

Atributo	Descripción	Elementos que pueden usarlo
onblur	Deseleccionar el elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
onclick	Pinchar y soltar el ratón	Todos los elementos
ondblclick	Pinchar dos veces seguidas con el ratón	Todos los elementos
onfocus	Seleccionar un elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla (sin soltar)	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	La página se ha cargado completamente	<body>
onmousedown	Pulsar (sin soltar) un botón del ratón	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos
onmouseout	El ratón *"sale"* del elemento (pasa por encima de otro elemento)	Todos los elementos
onmouseover	El ratón *"entra"* en el elemento (pasa por encima del elemento)	Todos los elementos
onmouseup	Soltar el botón que estaba pulsado en el ratón	Todos los elementos
onreset	Inicializar el formulario (borrar todos sus datos)	<form>
onresize	Se ha modificado el tamaño de la ventana del navegador	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página (por ejemplo, al cerrar el navegador)	<body>

2.3.3. Atributos de foco

Atributo	Descripción
accesskey = "letra"	Establece una tecla de acceso rápido a un elemento HTML
tabindex = "numero"	Establece la posición del elemento en el orden de tabulación de la página. Su valor debe estar comprendido entre 0 y 32.767
onfocus, onblur	Controlan los eventos JavaScript que se ejecutan cuando el elemento obtiene o pierde el foco

2.3.4. Atributos para internacionalización

Atributo	Descripción
lang = "codigo de idioma"	Indica el idioma del elemento mediante un código predefinido
xml:lang = "codigo de idioma"	Indica el idioma del elemento mediante un código predefinido
dir	Indica la dirección del texto (útil para los idiomas que escriben de derecha a izquierda)

2.4. Elementos HTML

El lenguaje HTML clasifica a todos los elementos en dos grupos:

- elementos **en línea** (*inline elements* en inglés)
- elementos de **bloque** (*block elements* en inglés).

La principal diferencia entre los dos tipos de elementos es la forma en la que ocupan el espacio disponible en la página. Los elementos de bloque siempre empiezan en una nueva línea y ocupan todo el espacio disponible hasta el final de la línea, aunque sus contenidos no lleguen hasta el final de la línea.

Por su parte, los elementos en línea sólo ocupan el espacio necesario para mostrar sus contenidos.

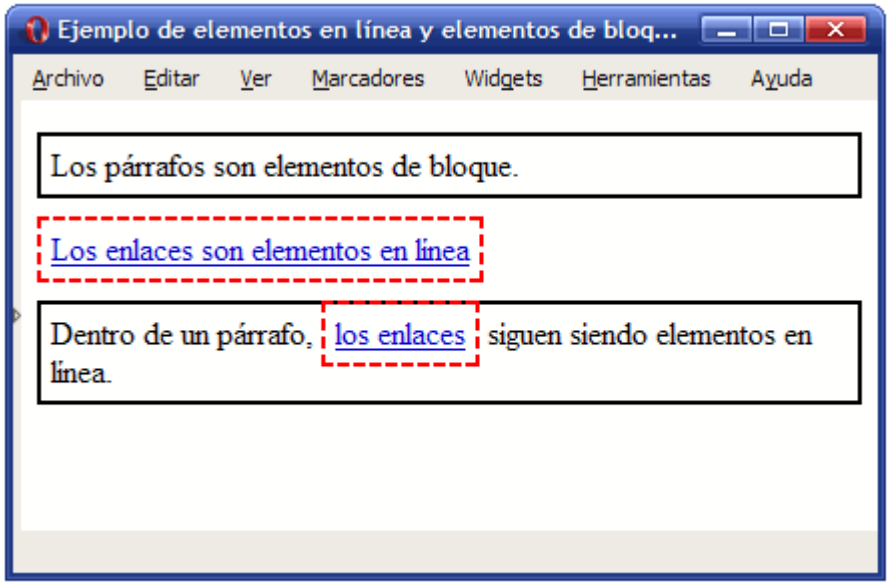


Figura 2.8 Diferencias entre elementos en línea y elementos de bloque

- Los elementos en línea definidos por HTML son: ***a, abbr, acronym, b, basefont, bdo, big, br, cite, code, dfn, em, font, i, img, input, kbd, label, q, s, samp, select, small, span, strike, strong, sub, sup, textarea, tt, u, var.***

- Los elementos de bloque definidos por HTML son: ***address, blockquote, center, dir, div, dl, fieldset, form, h1, h2, h3, h4, h5, h6, hr, isindex, menu, noframes, nos-cript, ol, p, pre, table, ul.***

- Los siguientes elementos también se considera que son de bloque: ***dd, dt, frame-set, li, tbody, td, tfoot, th, thead, tr.***

- Los siguientes elementos pueden ser en línea y de bloque según las circunstancias: ***button, del, iframe, ins, map, object, script.***

2.5. Sintaxis de las etiquetas XHTML

A continuación, se muestran las cinco restricciones básicas que introduce XHTML respecto a HTML en la sintaxis de sus etiquetas:

1) ***Las etiquetas se tienen que cerrar de acuerdo a como se abren:***

Ejemplo:

```
<p>Este es un párrafo con <a>un enlace</a></p>
```

2) ***Los nombres de las etiquetas y atributos siempre se escriben en minúsculas:***

Ejemplo:

```
<p>Este es un párrafo con <a href="http://www.google.com">un enlace</a></p>
```

3) ***El valor de los atributos siempre se encierra con comillas:***

Ejemplo:

```
<p>Este es un párrafo con <a href="http://www.google.com">un enlace</a></p>
```

4) ***Los atributos no se pueden comprimir:***

Ejemplo:

<dl compact="compact">...</dl>
(Este tipo de atributos en los que el nombre coincide con su valor no son muy habituales.)

5) ***Todas las etiquetas deben cerrarse siempre:***

La mayoría de las etiquetas HTML encierran un contenido de texto entre la etiqueta de apertura y la etiqueta de cierre. Sin embargo, algunas etiquetas especiales llamadas "etiquetas vacías" no necesitan encerrar ningún texto.

Por ejemplo, en lugar de abrir y cerrar de forma consecutiva la etiqueta (
</br>) se puede utilizar la sintaxis

Además de estas cinco restricciones básicas, XHTML incluye otros cambios más avanzados respecto a HTML:

- Antes de acceder al valor de un atributo, se eliminan todos los espacios en blanco que se encuentran antes y después del valor. Además, se eliminan todos los espacios en blanco sobrantes dentro del valor de un atributo. En otras palabras, si en el interior de un atributo se incluyen varios espacios en blanco seguidos, se eliminan todos salvo un único espacio en blanco utilizado para separar las diferentes palabras.

- Como se explicará más adelante al hablar de la etiqueta <script>, el código JavaScript debe encerrarse entre unas etiquetas especiales ***(<![CDATA[…]]>)*** para evitar que el navegador interprete de forma errónea caracteres como & y <.

- Las páginas XHTML deben prescindir del atributo name para identificar de forma única a los elementos. En su lugar, siempre debe utilizarse el atributo id. De hecho, en la versión 1.0 del estándar XHTML, el atributo name se ha declarado obsoleto para las etiquetas ***a, applet, form, frame, iframe, img y map.***

2.6. DOCTYPE

El estándar XHTML deriva de XML, por lo que comparte con el muchas de sus normas y sintaxis. Uno de los conceptos fundamentales de XML es la utilización del DTD o *Document Type Definition* (*"Definición del Tipo de Documento"*).

XHTML 1.0 Estricto

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Se trata de la variante con las normas más estrictas y las restricciones más severas. Las páginas web que incluyan este doctype, no pueden utilizar atributos relacionados con el aspecto de los contenidos, por lo que requiere una separación total de código HTML y estilos CSS.

XHTML 1.0 Transitorio

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Se trata de una variante menos estricta que la anterior, ya que permite el uso de algunos atributos HTML relacionados con el aspecto de los elementos.

XHTML 1.0 Frameset

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

El namespace que utilizan todas las páginas XHTML (independientemente de la versión y del DOCTYPE) es http://www.w3.org/1999/xhtml y se indica de la siguiente manera:

```
<html xmlns="http://www.w3.org/1999/xhtml"> ... </html>
```

De esta forma, es habitual que las páginas XHTML comiencen con el siguiente código:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="es" xml:lang="es">
```

Aunque el código anterior es mucho más complicado que una simple etiqueta <html>, es imprescindible para que las páginas XHTML creadas sean correctas y superen satisfactoriamente el proceso de validación que se muestra en los capítulos siguientes.

Ahora en HTML5 El encabezado o la primera línea se ha simplificado considerablemente, quedando de la siguiente manera.

```
<!DOCTYPE html>
```

3. Ejemplos de enlaces habituales

3.1 Enlace al inicio del sitio web

```
<a href="/">Inicio</a>
```

Al pulsar el enlace anterior desde cualquier página web, se vuelve directamente a la página de inicio, *home* o página principal del sitio web.

3.2. Enlace a un email

```
<a href="mailto:nombre@direccion.com" title="Dirección de email para solicitar más información"> Solicita más información </a>
```

3.3. Enlace a un archivo FTP

Para enlazar un archivo almacenado en un servidor FTP, la parte del protocolo de la URL debe cambiar de http:// a ftp://:

```
<a href="ftp://ftp.ejemplo.com/ruta/archivo.zip" title="Archivo comprimido de los contenidos"> Descarga un ZIP con todos los contenidos </a>
```

3.4. Enlazar varias hojas de estilos CSS

```
<link rel="stylesheet" type="text/css" href="/css/comun.css" /> <link rel="stylesheet" type="text/css" href="/css/secciones.css" />
```

3.5. Enlazar hojas de estilos CSS para diferentes medios

```
<link rel="stylesheet" type="text/css" href="/css/comun.css" media="screen, projection" />
<link rel="stylesheet" type="text/css" href="/css/impresora.css" media="print" />
<link rel="stylesheet" type="text/css" href="/css/movil.css" media="handheld" />
```

3.6. Enlazar el favicon

El *favicon* o icono para favoritos es el pequeño icono que muestran las páginas en varias partes del navegador.
Dependiendo del navegador que se utilice, este icono se muestra en la barra de direcciones, en la barra de título del navegador y/o en el menú de favoritos/marcadores.

```
<link rel="shortcut icon" href="/favicon.ico" type="image/ico" />
```

Etiqueta	****
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	En línea
Descripción	Realza con la máxima importancia el texto que encierra

Etiqueta	**<h1>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Define los títulos de las secciones de mayor importancia de la página.

Etiqueta	**<p>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Delimita el contenido de un párrafo de texto

Etiqueta	****
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	En línea
Descripción	Realza la importancia del texto que encierra

Etiqueta	**<ins>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `cite = "url"` - Indica la URL de la página en la que se puede obtener más información sobre el motivo por el que se realizó la modificación. • `datetime = "fecha"` - Especifica la fecha y hora en la que se realizó el cambio
Tipo de elemento	Bloque y en línea
Descripción	Se emplea para marcar una modificación en los contenidos originales consistente en la inserción de un nuevo contenido

Etiqueta	**<del>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `cite = "url"` - Indica la URL de la página en la que se puede obtener más información sobre el motivo por el que se realizó la modificación. • `datetime = "fecha"` - Especifica la fecha y hora en la que se realizó el cambio
Tipo de elemento	Bloque y en línea
Descripción	Se emplea para marcar una modificación en los contenidos originales consistente en el borrado de cierto contenido

Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `title = "texto"` - Indica el significado completo de la abreviatura
Tipo de elemento	En línea
Descripción	Se emplea para marcar las abreviaturas del texto y proporcionar el significado de esas abreviaturas

Etiqueta	**<acronym>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `title = "texto"` - Indica el significado completo del acrónimo o sigla
Tipo de elemento	En línea

Etiqueta	**<dfn>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `title = "texto"` - Indica el significado completo del término
Tipo de elemento	En línea
Descripción	Se emplea para marcar las definiciones de ciertos términos y proporcionar el significado de esos términos

Etiqueta	**<cite>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	En línea
Descripción	Se emplea para marcar una cita o una referencia a otras fuentes

Etiqueta	** **
Atributos comunes	básicos
Atributos propios	-
Tipo de elemento	En línea y etiqueta vacía
Descripción	Fuerza al navegador a insertar una nueva línea

Etiqueta	**<code>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	En línea
Descripción	Delimita el texto considerado un fragmento de código fuente

Etiqueta	**<a>**
Atributos comunes	básicos, internacionalización, eventos y foco
Atributos propios	• `name = "texto"` - Permite nombrar al enlace para que se pueda acceder desde otros enlaces • `href = "url"` - Indica la URL del recurso que se quiere enlazar
Tipo de elemento	En línea
Descripción	Se emplea para enlazar todo tipo de recursos

Etiqueta	**<script>**
Atributos comunes	-
Atributos propios	• `src = "url"` - Indica la dirección del archivo que contiene el código • `type = "tipo_de_contenido"` - Permite "avisar" al navegador sobre el tipo de código que se incluye (normalmente JavaScript) • `defer = "defer"` - El código no va a modificar el contenido de la página web • `charset = "tipo_de_charset"` - Describe la codificación del código enlazado
Tipo de elemento	Bloque y en línea (también puede ser una etiqueta vacía)
Descripción	Se emplea para enlazar o definir un bloque de código (normalmente JavaScript)

Etiqueta	**<link>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• Los siguientes con el mismo significado que para la etiqueta "a": charset, href, hreflang, type, rel y rev • `media = "tipo_de_medio"` - Indica el medio para el que debe aplicarse la relación
Tipo de elemento	Etiqueta vacía
Descripción	Se emplea para enlazar y establecer relaciones entre el documento y otros recursos

Etiqueta	**<ul>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para definir listas no ordenadas

Etiqueta	**<li>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para definir los elementos de las listas (ordenadas y no ordenadas)

Etiqueta	**<ol>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para definir listas ordenadas

Etiqueta	**<dl>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para definir listas de definición

Etiqueta	**<dt>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para definir los términos de los elementos de una lista de definición

Etiqueta	**<dd>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para indicar las definiciones de los elementos de una lista de definición

Etiqueta	**<img>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `src = "url"` - Indica la URL de la imagen que se muestra • `alt = "texto"` - Descripción corta de la imagen • `longdesc = "url"` - Indica una URL en la que puede encontrarse una descripción más detallada de la imagen • `name = "texto"` - Nombre del elemento imagen • `height = "unidad_de_medida"` - Indica la altura con la que se debe mostrar la imagen (no es obligatorio que coincida con la altura original de la imagen) • `width = "unidad_de_medida"` - Indica la anchura con la que se debe mostrar la imagen (no es obligatorio que coincida con la anchura original de la imagen)
Tipo de elemento	En línea y etiqueta vacía
Descripción	Se emplea para incluir imágenes en los documentos

Etiqueta	**<area>**
Atributos comunes	básicos, internacionalización, eventos y foco
Atributos propios	• `href = "url"` - URL a la que se accede al pinchar sobre el área • `nohref = "nohref"` - Se emplea para las áreas que no son seleccionables • `shape = "default | rect | circle | poly"` - Indica el tipo de área que se define (toda la imagen, rectangular, circular o poligonal) • `coords = "lista de números"` - Se trata de una lista de números separados por comas que representan las coordenadas del área. Rectangular = X1,Y1,X2,Y2 (coordenadas X e Y del vértice superior izquierdo y coordenadas X e Y del vértice inferior derecho). Circular = X1,Y1,R (coordenadas X e Y del centro y radio del círculo). Poligonal = X1,Y1,X2,Y2,...,XnYn (coordenadas de los vértices del polígono. Si las últimas coordenadas no son iguales que las primeras, se cierra automáticamente el polígono uniendo ambos vértices)
Tipo de elemento	Etiqueta vacía
Descripción	Se emplea para definir las distintas áreas que forman un mapa de imagen

Etiqueta	**<map>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `name = "texto"` - Nombre que identifica de forma única al mapa definido (es obligatorio indicar un nombre único)
Tipo de elemento	Bloque y en línea
Descripción	Se emplea para definir mapas de imagen

Etiqueta	**<table>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `summary = "texto"` - Permite describir el contenido de la tabla (lo utilizan los buscadores y las personas discapacitadas)
Tipo de elemento	Bloque
Descripción	Se emplea para definir tablas de datos

Etiqueta	**<tr>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para definir cada fila de las tablas de datos

Etiqueta	**<td>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `abbr = "texto"` - Permite describir el contenido de la celda (empleado sobre todo en los navegadores utilizados por personas discapacitadas) • `headers = "lista_de_id"` - Indica las celdas que actúan como cabeceras para esta celda (los títulos de las columnas y filas). Se indica como una lista de valores del atributo "id" de celdas • `scope = "col, row, colgroup, rowgroup"` - Indica las celdas para las que esta celda será su cabecera. Ej: scope="col" indica que esta celda es la cabecera de todas las demás celdas que están en la misma columna • `colspan = "numero"` - Número de columnas que ocupa esta celda • `rowspan = "numero"` - Número de filas que ocupa esta celda
Tipo de elemento	Bloque
Descripción	Se emplea para definir cada una de las celdas que forman las filas de una tabla, es decir, las columnas de la tabla

Etiqueta	**<th>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `abbr = "texto"` - Permite describir el contenido de la celda (empleado sobre todo en los navegadores utilizados por personas discapacitadas) • `headers = "lista_de_id"` - Indica las celdas que actúan como cabeceras para esta celda (los títulos de las columnas y filas). Se indica como una lista de ID de celdas • `scope = "col, row, colgroup, rowgroup"` - Indica las celdas para las que esta celda será su cabecera. Ej: scope="col" indica que esta celda es la cabecera de todas las demás celdas que están en la misma columna • `colspan = "numero"` - Número de columnas que ocupa esta celda • `rowspan = "numero"` - Número de filas que ocupa esta celda
Tipo de elemento	Bloque
Descripción	Se emplea para definir las celdas que son cabecera de una fila o de una columna de la tabla

Etiqueta	**<caption>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	En línea
Descripción	Se emplea para definir la leyenda o título de una tabla

Etiqueta	**<form>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `action = "url"` - Indica la URL que se encarga de procesar los datos del formulario • `method = "POST o GET"` - Método HTTP empleado al enviar el formulario • `enctype = "application/x-www-form-urlencoded o multipart/form-data"` - Tipo de codificación empleada al enviar el formulario al servidor (sólo se indica de forma explícita en los formularios que permiten adjuntar archivos) • `accept = "tipo_de_contenido"` - Lista separada por comas de todos los tipos de archivos aceptados por el servidor (sólo para los formularios que permiten adjuntar archivos) • Otros: accept-charset, onsubmit, onreset
Tipo de elemento	Bloque
Descripción	Se emplea para insertar un formulario en la página

Etiqueta	**<input>**
Atributos comunes	básicos, internacionalización, eventos y foco
Atributos propios	• `type = "text | password | checkbox | radio | submit | reset | file | hidden | image | button"` - Indica el tipo de control que se incluye en el formulario • `name = "texto"` - Asigna un nombre al control (es imprescindible para que el servidor pueda procesar el formulario) • `value = "texto"` - Valor inicial del control • `size = "unidad_de_medida"` - Tamaño inicial del control (para los campos de texto y de password se refiere al número de caracteres, en el resto de controles se refiere a su tamaño en píxel) • `maxlength = "numero"` - Máximo número de caracteres para los controles de texto y de password • `checked = "checked"` - Para los controles checkbox y radiobutton permite indicar qué opción aparece preseleccionada • `disabled = "disabled"` - El control aparece deshabilitado y su valor no se envía al servidor junto con el resto de datos • `readonly = "readonly"` - El contenido del control no se puede modificar • `src = "url"` - Para el control que permite crear botones con imágenes, indica la URL de la imagen que se emplea como botón de formulario • `alt = "texto"` - Descripción del control
Tipo de elemento	En línea y etiqueta vacía
Descripción	Se emplean para insertar un control en un formulario

Etiqueta	**<fieldset>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Se emplea para agrupar de forma lógica varios campos de un formulario

Etiqueta	**<legend>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	En línea
Descripción	Se emplea para definir el título o leyenda de un conjunto de campos de formulario agrupados con la etiqueta fieldset

Etiqueta	**<label>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `for = "id_de_elemento"` - Indica el ID del campo del formulario para el que este elemento es su título • Otros: accesskey, onfocus y onblur
Tipo de elemento	En línea
Descripción	Se emplea para definir el título o leyenda de los campos definidos en un formulario

Etiqueta	**<textarea>**
Atributos comunes	básicos, internacionalización, eventos y foco
Atributos propios	• `rows = "numero"` - Número de filas de texto que mostrará el textarea • `cols = "numero"` - Número de caracteres que se muestran en cada fila del textarea • Otros: name, disabled, readonly, onselect, onchange, onfocus, onblur
Tipo de elemento	En línea
Descripción	Se emplea para incluir un área de texto en un formulario

Etiqueta	**<select>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `size = "numero"` - Número de filas que se muestran de la lista (por defecto sólo se muestra una) • `multiple = "multiple"` - Si se incluye, se permite seleccionar más de un elemento • Otros: name, disabled, onchange, onfocus, onblur
Tipo de elemento	En línea
Descripción	Se emplea para incluir una lista desplegable en un formulario

Etiqueta	**<option>**
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	• `selected = "selected"` - Indica si el elemento aparece seleccionado por defecto al cargarse la página • `value = "texto"` - El valor que se envía al servidor cuando el usuario elige esa opción • Otros: label, disabled
Tipo de elemento	-
Descripción	Se emplea para definir cada elemento de una lista desplegable

Etiqueta	** **
Atributos comunes	básicos, internacionalización y eventos
Atributos propios	-
Tipo de elemento	Bloque
Descripción	Agrupa elementos de bloque

Etiqueta	**<head>**
Atributos comunes	internacionalización
Atributos propios	• `profile = "url"` - Especifica la URL del perfil o perfiles que utilizan los metadatos • `lang = "codigo_de_idioma"` - Especifica el idioma principal de los contenidos de la página
Tipo de elemento	-
Descripción	Define la cabecera del documento HTML

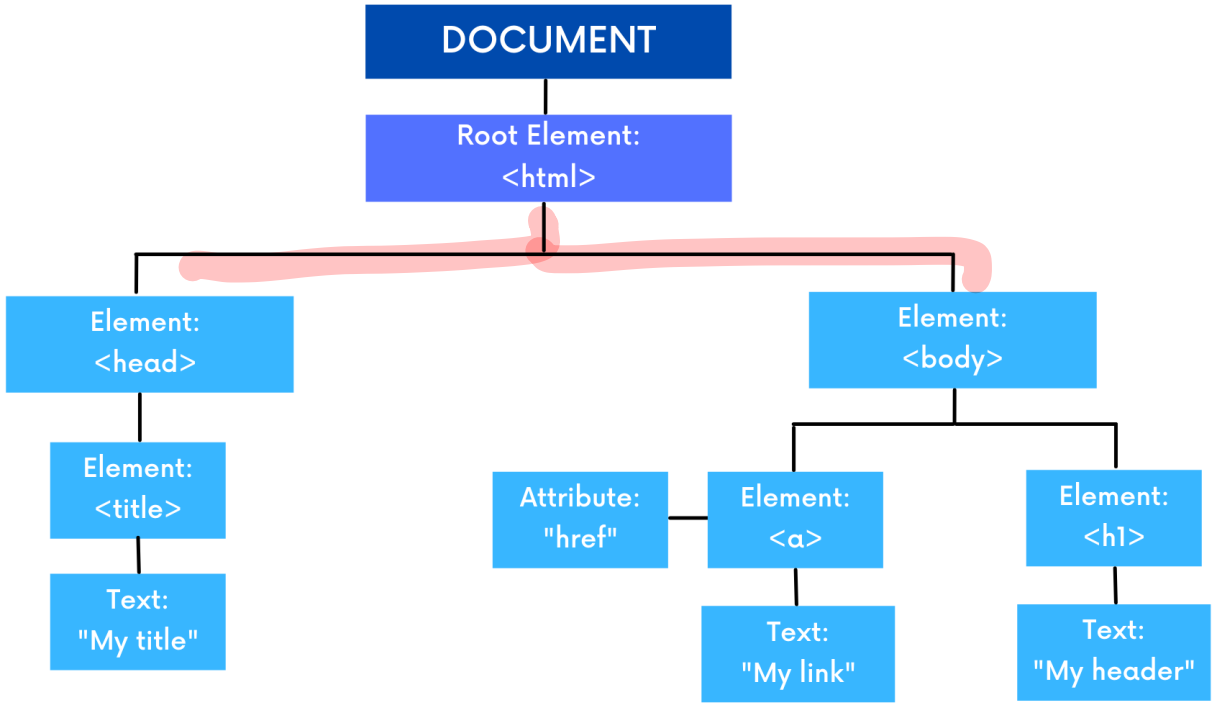
Etiqueta	**<title>**
Atributos comunes	internacionalización
Atributos propios	• `lang = "codigo_de_idioma"` - Especifica el idioma principal del título de la página
Tipo de elemento	-
Descripción	Define el título del documento HTML

Etiqueta	**<meta>**
Atributos comunes	internacionalización
Atributos propios	• `name = "texto"` - El nombre de la propiedad que se define (no existe una lista oficial de propiedades) • `content = "texto"` - El valor de la propiedad definida (no existe una lista de valores permitidos) • `http-equiv = "texto"` - En ocasiones, reemplaza al atributo "name" y lo emplean los servidores para adaptar sus respuestas al documento • `scheme = "texto"` - Indica el esquema que se debe emplear para interpretar el valor de la propiedad
Tipo de elemento	-
Descripción	Permite definir el valor de los metadatos que forman la metainformación del documento

EL DOM

El árbol de salida ("árbol de análisis") está formado por elementos DOM y nodos de atributo. DOM son las siglas de "Document Object Model" (modelo de objetos del documento). Es la presentación de objetos del documento HTML y la interfaz de los elementos HTML para el mundo exterior, como JavaScript.

La raíz del árbol es el objeto ***document***



El modelo DOM guarda una relación casi de uno a uno con el marcado. Veamos un ejemplo de marcado:

```
<html>
  <body>
      <p> Hello World </p>
      <div> <img src="example.png"/></div>
  </body>
</html>
```

El marcado anterior se traduciría en el siguiente árbol de DOM:

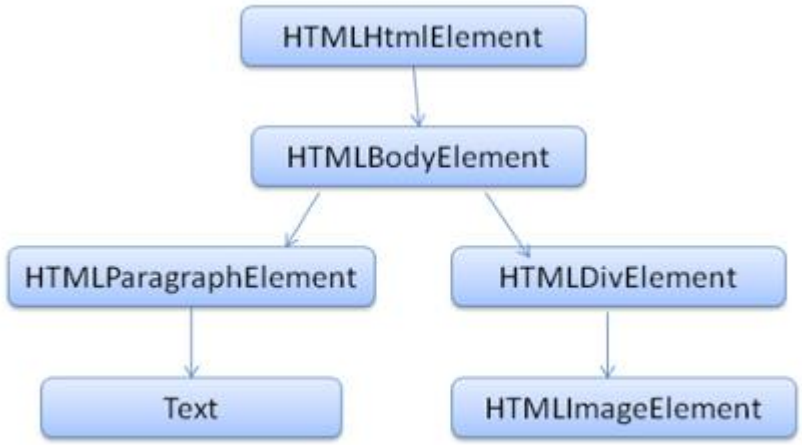
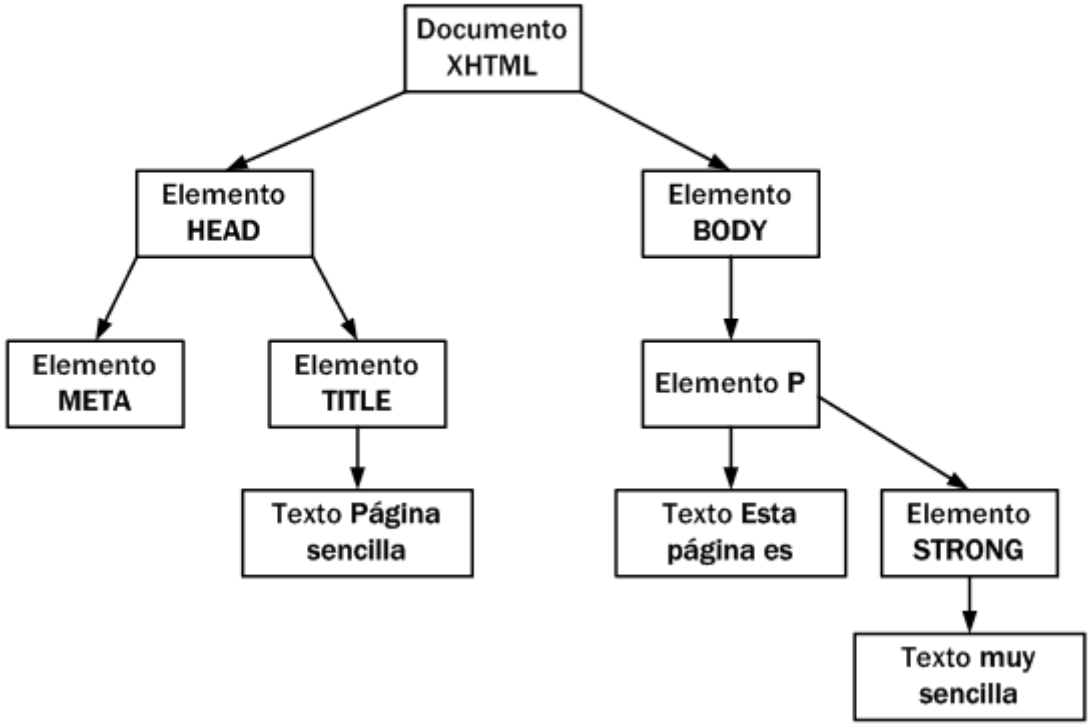


Figura : árbol de DOM del marcado de ejemplo

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

 <html xmlns="http://www.w3.org/1999/xhtml">
                <head> <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
                        <title>Página sencilla</title>
                </head>
                <body>
                                <p>Esta página es <strong>muy sencilla</strong></p>
                 </body>
</html>
```

Se transforma en el siguiente árbol de nodos:



- La creación del *Document Object Model* o **DOM** es una de las innovaciones que más ha influido en el desarrollo de las páginas web dinámicas y de las aplicaciones web más complejas.

- DOM permite a los programadores web acceder y manipular las páginas XHTML como si fueran documentos XML. De hecho, DOM se diseñó originalmente para manipular de forma sencilla los documentos XML.

- A pesar de sus orígenes, DOM se ha convertido en una utilidad disponible para la mayoría de los lenguajes de programación (Java, PHP, JavaScript) y cuyas únicas diferencias se encuentran en la forma de implementarlo.

Tipos de nodos

La especificación completa de DOM define 12 tipos de nodos, aunque las páginas XHTML habituales se pueden manipular manejando solamente cuatro o cinco tipos de nodos:

- Document, nodo raíz del que derivan todos los demás nodos del árbol.
- Element, representa cada una de las etiquetas HTML. Se trata del único nodo que puede contener atributos y el único del que pueden derivar otros nodos.
- Attr, se define un nodo de este tipo para representar cada uno de los atributos de las etiquetas HTML, es decir, uno por cada par atributo=valor.
- Text, nodo que contiene el texto encerrado por una etiqueta HTML.
- Comment, representa los comentarios incluidos en la página HTML.

Los otros tipos de nodos existentes que no se van a considerar son DocumentType, CDataSection, DocumentFragment, Entity, EntityReference, ProcessingInstruction y Notation.

Acceso directo a los nodos a través de JavaScript

Una vez construido automáticamente el árbol completo de nodos DOM, ya es posible utilizar las funciones DOM para acceder de forma directa a cualquier nodo del árbol. Como acceder a un nodo del árbol es equivalente a acceder a "*un trozo*" de la página, una vez construido el árbol, ya es posible manipular de forma sencilla la página: acceder al valor de un elemento, establecer el valor de un elemento, mover un elemento de la página, crear y añadir nuevos elementos, etc.

getElementsByTagName()

Como sucede con todas las funciones que proporciona DOM, la función getElementsByTagName() tiene un nombre muy largo, pero que lo hace autoexplicativo.

La función getElementsByTagName(nombre Etiqueta) obtiene todos los elementos de la página HTML cuya etiqueta sea igual que el parámetro que se le pasa a la función.

El siguiente ejemplo muestra cómo obtener todos los párrafos de una página HTML:

```
var parrafos = document.getElementsByTagName("p");
```

El valor que devuelve la función es un array con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. El valor devuelto es un array de nodos DOM, no un array de cadenas de texto o un array de objetos normales.

De este modo, se puede obtener el primer párrafo de la página de la siguiente manera:

```
var primerParrafo = parrafos[0];
```

De la misma forma, se podrían recorrer todos los párrafos de la página con el siguiente código:

```
for(var i=0; i< parrafos.length; i++) {
  var parrafo = parrafos[i];
}
```

La función ***getElementsByTagName()*** se puede aplicar de forma recursiva sobre cada uno de los nodos devueltos por la función. En el siguiente ejemplo, se obtienen todos los enlaces del primer párrafo de la página:

```
var parrafos = document.getElementsByTagName("p");
var primerParrafo = parrafos[0];
var enlaces = primerParrafo.getElementsByTagName("a");
```

getElementsByName()

La función **getElementsByName()** es similar a la anterior, pero en este caso se buscan los elementos cuyo atributo name sea igual al parámetro proporcionado. En el siguiente ejemplo, se obtiene directamente el único párrafo con el nombre indicado:

```
var parrafoEspecial = document.getElementsByName("especial");
```

```
<p name="prueba">...</p>
<p name="especial">...</p>
<p>...</p>
```

Normalmente el atributo name es único para los elementos HTML que lo definen, por lo que es un método muy práctico para acceder directamente al nodo deseado. En el caso de los elementos HTML radiobutton, el atributo name es común a todos los radiobutton que están relacionados, por lo que la función devuelve una colección de elementos.

Creación de elementos HTML

Como se ha visto, un elemento HTML sencillo, como por ejemplo un párrafo, genera dos nodos: el primer nodo es de tipo Element y representa la ***etiqueta*** <p> y el segundo nodo es de tipo Text y representa el ***contenido textual*** de la etiqueta <p>.

Por este motivo, crear y añadir a la página un nuevo elemento HTML sencillo consta de cuatro pasos diferentes:

- Creación de un nodo de tipo Element que represente al elemento.
- Creación de un nodo de tipo Text que represente el contenido del elemento.
- Añadir el nodo Text como nodo hijo del nodo Element.
- Añadir el nodo Element a la página, en forma de nodo hijo del nodo correspondiente al lugar en el que se quiere insertar el elemento.

getElementById()

La función ***getElementById()*** es la más utilizada cuando se desarrollan aplicaciones web dinámicas. Se trata de la función preferida para acceder directamente a un nodo y poder leer o modificar sus propiedades.

La función ***getElementById()*** devuelve el elemento HTML cuyo atributo id coincide con el parámetro indicado en la función. Como el atributo id debe ser único para cada elemento de una misma página, la función devuelve únicamente el nodo deseado.

```
var cabecera = document.getElementById("cabecera");
```

```
<div id="cabecera">
  <a href="/" id="logo">...</a>
</div>
```

```
// Crear nodo de tipo Element
var parrafo = document.createElement("p");

// Crear nodo de tipo Text
var contenido = document.createTextNode(¡"Hola Mundo!");

// Añadir el nodo Text como hijo del nodo Element
parrafo.appendChild(contenido);

// Añadir el nodo Element como hijo de la pagina
document.body.appendChild(parrafo);
```

El proceso de creación de nuevos nodos puede llegar a ser tedioso, ya que implica la utilización de tres funciones DOM:

createElement(etiqueta): crea un nodo de tipo Element que representa al elemento HTML cuya etiqueta se pasa como parámetro.
createTextNode(contenido): crea un nodo de tipo Text que almacena el contenido textual de los elementos HTML.

nodoPadre.appendChild(nodoHijo): añade un nodo como hijo de otro nodo. Se debe utilizar al menos dos veces con los nodos habituales: en primer lugar, se añade el nodo Text como hijo del nodo Element y a continuación se añade el nodo Element como hijo de algún nodo de la página.

Eliminación de nodos

Afortunadamente, eliminar un nodo del árbol DOM de la página es mucho más sencillo que añadirlo. En este caso, solamente es necesario utilizar la función ***removeChild()***:

```
var parrafo = document.getElementById("provisional");
parrafo.parentNode.removeChild(parrafo);

<p id="provisional">...</p>
```

Acceso directo a los atributos HTML

Una vez que se ha accedido a un nodo, el siguiente paso natural consiste en acceder y/o modificar sus atributos y propiedades. Mediante DOM, es posible acceder de forma sencilla a todos los atributos HTML y todas las propiedades CSS de cualquier elemento de la página.

Los atributos HTML de los elementos de la página se transforman automáticamente en propiedades de los nodos. Para acceder a su valor, simplemente se indica el nombre del atributo HTML detrás del nombre del nodo.

El siguiente ejemplo obtiene de forma directa la dirección a la que enlaza el enlace:

```
var enlace = document.getElementById("enlace");
alert(enlace.href); // muestra http://www...com
console.log
<a id="enlace" href="http://www...com">Enlace</a>
```

Las propiedades CSS no son tan fáciles de obtener como los atributos HTML.

Para obtener el valor de cualquier propiedad CSS del nodo, se debe utilizar el atributo ***style***. El siguiente ejemplo obtiene el valor de la propiedad margin de la imagen:

```
var imagen = document.getElementById("imagen");
alert(imagen.style.margin);

<img id="imagen" style="margin:0; border:0;" src="logo.png" />
```

Si el nombre de una propiedad CSS es compuesto, se accede a su valor modificando ligeramente su nombre:

```
var parrafo = document.getElementById("parrafo");
alert(parrafo.style.fontWeight);  // muestra "bold"

<p id="parrafo" style="font-weight: bold;">...</p>
```

La transformación del nombre de las propiedades CSS compuestas consiste en eliminar todos los guiones medios (-) y escribir en mayúscula la letra siguiente a cada guion medio. A continuación, se muestran algunos ejemplos:

- font-weight se transforma en **fontWeight**
- line-height se transforma en **lineHeight**
- border-top-style se transforma en **borderTopStyle**
- list-style-image se transforma en **listStyleImage**

camelCase

El único atributo HTML que no tiene el mismo nombre en HTML y en las propiedades DOM es el atributo ***class***. Como la palabra class está reservada por JavaScript, no es posible utilizarla para acceder al atributo class del elemento HTML.

En su lugar, DOM utiliza el nombre ***className*** para acceder al atributo class de HTML:

```
var parrafo = document.getElementById("parrafo");
alert(parrafo.class); // muestra "undefined"
alert(parrafo.className); // muestra "normal"

<p id="parrafo" class="normal">...</p>
```

Leer la propiedad de un elemento mediante ***innerHTML***

Para obtener el marcado HTML dentro de un elemento, se puede utilizar la siguiente sintaxis

```
let content = element.innerHTML;
```

Ejemplo.-

```
<ul id="menu">
   <li>Home</li>
   <li>Services</li>
</ul>
```

para obtener el contenido de <ul> utilizariamos

```
let menu = document.getElementById('menu');
console.log(menu.innerHTML);
```

Ejercicio 1

A partir de la página web proporcionada (Ejercicio 1) y utilizando las funciones DOM, mostrar por pantalla la siguiente información:

1. Número de enlaces de la página
2. Dirección a la que enlaza el penúltimo enlace
3. Numero de enlaces que enlazan a http://prueba
4. Número de enlaces del tercer párrafo

Ejercicio 2

Completar el código JavaScript proporcionado para que cuando se pinche sobre el enlace se muestre completo el contenido de texto. Además, el enlace debe dejar de mostrarse después de pulsarlo por primera vez. La acción de pinchar sobre un enlace forma parte de los "***Eventos***" de JavaScript. En este ejercicio, sólo se debe saber que, al pinchar sobre el enlace, se ejecuta la función llamada ***muestra***().

Ejercicio 3

Completar el código JavaScript proporcionado para que se añadan nuevos elementos a la lista cada vez que se pulsa sobre el botón. Utilizar las funciones DOM para crear nuevos nodos y añadirlos a la lista existente. Al igual que sucede en el ejercicio anterior, la acción de pinchar sobre un botón forma parte de los "Eventos" de JavaScript. En este ejercicio, sólo se debe saber que, al pinchar sobre el botón, se ejecuta la función llamada ***anade***().

CSS
HTML

1.2. Breve historia de CSS

- Las hojas de estilos aparecieron poco después que el lenguaje de etiquetas SGML, alrededor del año 1970. Desde la creación de SGML, se observó la necesidad de definir un mecanismo que permitiera aplicar de forma consistente diferentes estilos a los documentos electrónicos.

- En 1995, el W3C decidió apostar por el desarrollo y estandarización de CSS y lo añadió a su grupo de trabajo de HTML. A finales de 1996, el W3C publicó la primera recomendación oficial, conocida como "CSS nivel 1".

- A principios de 1997, el W3C decide separar los trabajos del grupo de HTML en tres secciones: el grupo de trabajo de HTML, el grupo de trabajo de DOM y el grupo de trabajo de CSS.

1.5. Funcionamiento básico de CSS

Antes de que se generalizara el uso de CSS, los diseñadores de páginas web utilizaban etiquetas HTML especiales para modificar el aspecto de los elementos de la página. El siguiente ejemplo muestra una página HTML con estilos definidos sin utilizar CSS:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
     <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
     <title>Ejemplo de estilos sin CSS</title>
  </head>
  <body>
      <h1><font color="red" face="Arial" size="5">Titular de la página</font></h1>
      <p><font color="gray" face="Verdana" size="2">Un párrafo de texto no muy largo.</font></p>
  </body>
</html>
```

```
<style type="text/css">
  h1 { color: red; font-family: Arial; font-size: large; }
  p { color: gray; font-family: Verdana; font-size: medium; }
</style>
```

1.6. Cómo incluir CSS en un documento XHTML

1.6.1. Incluir CSS en el mismo documento HTML

Los estilos se definen en una zona específica del propio documento HTML. Se emplea la etiqueta <style> de HTML y solamente se pueden incluir en la cabecera del documento (sólo dentro de la sección <head>).

Ejemplo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
    <title>Ejemplo de estilos CSS en el propio documento</title>
  <style type="text/css">
    p { color: black; font-family: Verdana; }
  </style>
  </head>
  <body>
    <p>Un párrafo de texto.</p>
  </body>
</html>
```

1.6.2. Definir CSS en un archivo externo

En este caso, todos los estilos CSS se incluyen en un archivo de tipo CSS que las páginas HTML enlazan mediante la etiqueta <link>. Un archivo de tipo CSS no es más que un archivo simple de texto cuya extensión es .css Se pueden crear todos los archivos CSS que sean necesarios y cada página HTML puede enlazar tantos archivos CSS como necesite.

Si se quieren incluir los estilos del ejemplo anterior en un archivo CSS externo, se deben seguir los siguientes pasos:

1) Se crea un archivo de texto y se le añade solamente el siguiente contenido:

p { color: black; font-family: Verdana; }

2) Se guarda el archivo de texto con el nombre estilos.css Se debe poner especial atención a que el archivo tenga extensión ***.css*** y no .txt

3) En la página HTML se enlaza el archivo CSS externo mediante la etiqueta ***<link>***:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml">
  <head>
     <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
     <title>Ejemplo de estilos CSS en un archivo externo</title>
     <link rel="stylesheet" type="text/css" href="/css/estilos.css" media="screen" />
  </head>
  <body>
     <p>Un párrafo de texto.</p>
  </body>
  </html>
```

Normalmente, la etiqueta <link> incluye cuatro atributos cuando enlaza un archivo CSS:

•***rel***: indica el tipo de relación que existe entre el recurso enlazado (en este caso, el archivo CSS) y la página HTML. Para los archivos CSS, siempre se utiliza el valor stylesheet

•***type***: indica el tipo de recurso enlazado. Sus valores están estandarizados y para los archivos CSS su valor siempre es text/css

•***href***: indica la URL del archivo CSS que contiene los estilos. La URL indicada puede ser relativa o absoluta y puede apuntar a un recurso interno o externo al sitio web.

•***media***: indica el medio en el que se van a aplicar los estilos del archivo CSS. Más adelante se explican en detalle los medios CSS y su funcionamiento.

Aunque generalmente se emplea la etiqueta ***<link>*** para enlazar los archivos CSS externos, también se puede utilizar la etiqueta ***<style>***. La forma alternativa de incluir un archivo CSS externo se muestra a continuación:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 <html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
      <title>Ejemplo de estilos CSS en un archivo externo</title>
      <style type="text/css" media="screen">
        @import '/css/estilos.css';
      </style>
   </head>
   <body>
              <p>Un párrafo de texto.</p>
   </body>
</html>
```

- En este caso, para incluir en la página HTML los estilos definidos en archivos CSS externos se utiliza una regla especial de tipo ***@import***.

- Las reglas de tipo @import siempre preceden a cualquier otra regla CSS (con la única excepción de la regla @charset).

- La URL del archivo CSS externo se indica mediante una cadena de texto encerrada con comillas simples o dobles o mediante la palabra reservada url().

De esta forma, las siguientes reglas @import son equivalentes:

```
@import '/css/estilos.css’;
@import "/css/estilos.css";
@import url('/css/estilos.css’);
@import url("/css/estilos.css");
```

1.6.3. Incluir CSS en los elementos HTML

- El último método para incluir estilos CSS en documentos HTML es el peor y el menos utilizado, ya que tiene los mismos problemas que la utilización de las etiquetas <font>.

Ejemplo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
      <title>Ejemplo de estilos CSS en el propio documento</title>
   </head>
   <body>
         <p style="color: black; font-family: Verdana;">Un párrafo de texto.</p>
   </body>
 </html>
```

- Esta forma de incluir CSS directamente en los elementos HTML solamente se utiliza en determinadas situaciones en las que se debe incluir un estilo muy específico para un solo elemento concreto.

CSS define una serie de términos que permiten describir cada una de las partes que componen los estilos CSS. El siguiente esquema muestra las partes que forman un estilo CSS muy básico:

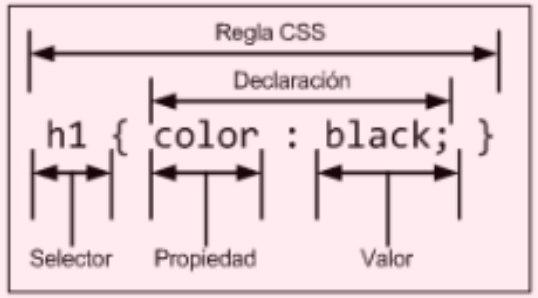


Figura 1.1 Componentes de un estilo CSS básico

Los diferentes términos se definen a continuación:

•***Regla***: cada uno de los estilos que componen una hoja de estilos CSS. Cada regla está compuesta de una parte de *"selectores"*, un símbolo de *"llave de apertura"* ({), otra parte denominada *"declaración"* y por último, un símbolo de *"llave de cierre"* (}).

•***Selector***: indica el elemento o elementos HTML a los que se aplica la regla CSS.

•***Declaración***: especifica los estilos que se aplican a los elementos. Está compuesta por una o más propiedades CSS.

•***Propiedad***: característica que se modifica en el elemento seleccionado, como por ejemplo su tamaño de letra, su color de fondo, etc.

•***Valor***: establece el nuevo valor de la característica modificada en el elemento.

1.8. Medios CSS

Una de las características más importantes de las hojas de estilos CSS es que permiten definir diferentes estilos para diferentes medios o dispositivos: pantallas, impresoras, móviles, proyectores, etc.

Medio	Descripción
all	Todos los medios definidos
braille	Dispositivos táctiles que emplean el sistema braille
embosed	Impresoras braille
handheld	Dispositivos de mano: móviles, PDA, etc.
print	Impresoras y navegadores en el modo *"Vista Previa para Imprimir"*
projection	Proyectores y dispositivos para presentaciones
screen	Pantallas de ordenador
speech	Sintetizadores para navegadores de voz utilizados por personas discapacitadas
tty	Dispositivos textuales limitados como teletipos y terminales de texto
tv	Televisores y dispositivos con resolución baja

1.8.1. Medios definidos con las reglas de tipo @media

```
@media print {
        body { font-size: 10pt } }

@media screen {
        body { font-size: 13px } }

@media screen, print {
        body { line-height: 1.2 } }
```

1.8.2. Medios definidos con las reglas de tipo @import

```
@import url("estilos_basicos.css") screen;
@import url("estilos_impresora.css") print;
```

Si no se indica el medio en una regla de tipo @import, el navegador sobreentiende que el medio es all, es decir, que los estilos se aplican en todos los medios.

1.8.3. Medios definidos con la etiqueta <link>

```
<link rel="stylesheet" type="text/css" media="screen" href="basico.css" />
<link rel="stylesheet" type="text/css" media="print, handheld" href="especial.css" />
```

Si la etiqueta <link> no indica el medio CSS, se sobreentiende que los estilos se deben aplicar a todos los medios, por lo que es equivalente a indicar media="all".

1.8.4. Medios definidos mezclando varios métodos

CSS también permite mezclar los tres métodos anteriores para indicar los medios en los que se aplica cada archivo CSS externo:

```
<link rel="stylesheet" type="text/css" media="screen" href="basico.css" />
@import url("estilos_seccion.css") screen;
@media print { /* Estilos específicos para impresora */ }
```

Capítulo 2. Selectores

2.1.1. Selector universal

Se utiliza para seleccionar todos los elementos de la página. El siguiente ejemplo elimina el margen y el relleno de todos los elementos HTML (por ahora no es importante fijarse en la parte de la declaración de la regla CSS):

```
* { margin: 0; padding: 0; }
```

2.1.2. Selector de tipo o etiqueta

Selecciona todos los elementos de la página cuya etiqueta HTML coincide con el valor del selector. El siguiente ejemplo selecciona todos los párrafos de la página:

```
p { ... }
```

2.1.3. Selector descendente

Selecciona los elementos que se encuentran dentro de otros elementos. Un elemento es descendiente de otro cuando se encuentra entre las etiquetas de apertura y de cierre del otro elemento.

El selector del siguiente ejemplo selecciona todos los elementos ** de la página que se encuentren dentro de un elemento *<p>*:

```
p span { color: red; }
```

¿De qué color se visualiza el texto?

```
p span { color: red; }
h1 span { color: blue; }
```

```
<body>
    <p><span>Un párrafo de texto.</span></p>
        <span><h1>otro texto</h1></span>
</body>
```

2.1.4. Selector de clase

Si se considera el siguiente código HTML de ejemplo:

```
<style>
 .destacado { color: red; }
</style>
<body>
    <p class="destacado">Lorem ipsum dolor sit amet...</p>
    <p>Nunc sed lacus et est adipiscing accumsan...</p>
    <p>Class aptent taciti sociosqu ad litora...</p>
</body>
```

Este tipo de selectores se llaman selectores de clase y son los más utilizados junto con los selectores de ID que se verán a continuación. La principal característica de este selector es que en una misma página HTML varios elementos diferentes pueden utilizar el mismo valor en el atributo class.

2.1.5. Selectores de ID

En ocasiones, es necesario aplicar estilos CSS a un único elemento de la página. Aunque puede utilizarse un selector de clase para aplicar estilos a un único elemento, existe otro selector más eficiente en este caso.

El selector de ID permite seleccionar un elemento de la página a través del valor de su atributo id. Este tipo de selectores sólo seleccionan un elemento de la página porque el valor del atributo id no se puede repetir en dos elementos diferentes de una misma página.

```
#destacado { color: red; }
```

```
<p>Primer párrafo</p>
<p id="destacado">Segundo párrafo</p>
<p>Tercer párrafo</p>
```

Al igual que los selectores de clase, en este caso también se puede restringir el alcance del selector mediante la combinación con otros selectores.

El siguiente ejemplo aplica la regla CSS solamente al elemento de tipo <p> que tenga un atributo id igual al indicado:

```
p#aviso { color: blue; }
```

2.2.1. Selector de hijos

Se trata de un selector similar al selector descendente, pero muy diferente en su funcionamiento. Se utiliza para seleccionar un elemento que es hijo directo de otro elemento y se indica mediante el "signo de mayor que" (>):

```
p > span { color: blue; }

<p><span>Texto1</span></p>
<p><a href="#"><span>Texto2</span></a></p>
```

- En el ejemplo anterior, el selector p > span se interpreta como "cualquier elemento que sea hijo directo de un elemento <p>", por lo que el primer elemento cumple la condición del selector.

- Sin embargo, el segundo elemento no la cumple porque es descendiente pero no es hijo directo de un elemento <p>.

2.2.2. Selector adyacente

El selector adyacente se emplea para seleccionar elementos que en el código HTML de la página se encuentran justo a continuación de otros elementos. Su sintaxis emplea el signo + para separar los dos elementos:

```
<body>
  <h1>Titulo1</h1>

  <h2>Subtítulo</h2> ...
  <h2>Otro subtítulo</h2> ...

</body>
```

```
h2 { color: green; }

h1 + h2 { color: red }
```

Las reglas CSS anteriores hacen que todos los <h2> de la página se vean de color verde, salvo aquellos <h2> que se encuentran inmediatamente después de cualquier elemento <h1> y que se muestran de color rojo.

2.2.3. Selector de atributos

El último tipo de selectores avanzados lo forman los selectores de atributos, que permiten seleccionar elementos HTML en función de sus atributos y/o valores de esos atributos.

Los cuatro tipos de selectores de atributos son:

- [nombre_atributo], selecciona los elementos que tienen establecido el atributo llamado nombre_atributo, independientemente de su valor.

- [nombre_atributo=valor], selecciona los elementos que tienen establecido un atributo llamado nombre_atributo con un valor igual a valor.

- [nombre_atributo~=valor], selecciona los elementos que tienen establecido un atributo llamado nombre_atributo y al menos uno de los valores del atributo es valor.

- [nombre_atributo|=valor], selecciona los elementos que tienen establecido un atributo llamado nombre_atributo y cuyo valor es una serie de palabras separadas con guiones, pero que comienza con valor. Este tipo de selector sólo es útil para los atributos de tipo lang que indican el idioma del contenido del elemento.

A continuación, se muestran algunos ejemplos de estos tipos de selectores:

```
/* Se muestran de color azul todos los enlaces que tengan un atributo "class", independientemente de su valor */
a[class] { color: blue; }

/* Se muestran de color azul todos los enlaces que tengan un atributo "class" con el valor "externo" */
a[class="externo"] { color: blue; }

/* Se muestran de color azul todos los enlaces que apunten al sitio "http://www.ejemplo.com" */
a[href="http://www.ejemplo.com"] { color: blue; }

/* Se muestran de color azul todos los enlaces que tengan un atributo "class" en el que al menos uno de sus valores sea "externo" */
a[class~="externo"] { color: blue; }

/* Selecciona todos los elementos de la página cuyo atributo "lang" sea igual a "en", es decir, todos los elementos en inglés */
*[lang=en] { ... }

/* Selecciona todos los elementos de la página cuyo atributo "lang" empiece por "es", es decir, "es", "es-ES", "es-AR", etc. */
*[lang|="es"] { color : red }
```

2.4. Herencia

Una de las características principales de CSS es la herencia de los estilos definidos para los elementos. Cuando se establece el valor de una propiedad CSS en un elemento, sus elementos descendientes heredan de forma automática el valor de esa propiedad. Si se considera el siguiente ejemplo:

Aunque la herencia de estilos se aplica automáticamente, se puede anular su efecto estableciendo de forma explícita otro valor para la propiedad que se hereda, como se muestra en el siguiente ejemplo:

```
<style type="text/css">
body { font-family: Arial; color: black; }
h1 { font-family: Verdana; }
p { color: red; }
</style>
<body>
<h1>Titular de la página</h1>
<p>Un párrafo de texto no muy largo.</p>
</body>
```

En el ejemplo anterior, se establece en primer lugar el color y tipo de letra del elemento <body>, por lo que todos los elementos de la página se mostrarían con ese mismo color y tipo de letra. No obstante, las otras reglas CSS modifican alguno de los estilos heredados.

De esta forma, los elementos <h1> de la página se muestran con el tipo de letra Verdana establecido por el selector h1 y se muestran de color negro que es el valor heredado del elemento <body>. Igualmente, los elementos <p> de la página se muestran del color rojo establecido por el selector p y con un tipo de letra Arial heredado del elemento <body>.

2.5. Colisiones de estilos

En las hojas de estilos complejas, es habitual que varias reglas CSS se apliquen a un mismo elemento HTML. El problema de estas reglas múltiples es que se pueden dar colisiones como la del siguiente ejemplo:

```
p { color: red; }
p { color: blue; }

<p>...</p>
```

El método seguido por CSS para resolver las colisiones de estilos se muestra a continuación:

1. Determinar todas las declaraciones que se aplican al elemento para el medio CSS seleccionado.
2. Ordenar las declaraciones según su origen (CSS de navegador, de usuario o de diseñador) y su prioridad.
3. Ordenar las declaraciones según lo específico que sea el selector. Cuanto más genérico es un selector, menos importancia tienen sus declaraciones.
4. Si después de aplicar las normas anteriores existen dos o más reglas con la misma prioridad, se aplica la que se indicó en último lugar.

Capítulo 3. Unidades de medida y colores

3.1.1. Unidades absolutas

Una medida indicada mediante unidades absolutas está completamente definida, ya que su valor no depende de otro valor de referencia. A continuación, se muestra la lista completa de unidades absolutas definidas por CSS y su significado:

- in, pulgadas (*"inches"*, en inglés). Una pulgada equivale a 2.54 centímetros.
- cm, centímetros.
- mm, milímetros.
- pt, puntos. Un punto equivale a 1 pulgada/72, es decir, unos 0.35 milímetros.
- pc, picas. Una pica equivale a 12 puntos, es decir, unos 4.23 milímetros.

```
body { margin: 0.5in; }
h1 { line-height: 2cm; }
p { word-spacing: 4mm; }
a { font-size: 12pt }
span { font-size: 1pc }
```

3.1.2. Unidades relativas

A continuación, se muestran las tres unidades de medida relativas definidas por CSS y la referencia que toma cada una para determinar su valor real:

- em, (no confundir con la etiqueta de HTML) relativa respecto del tamaño de letra del elemento.
- ex, relativa respecto de la altura de la letra x (*"equis minúscula"*) del tipo y tamaño de letra del elemento.
- px, (píxel) relativa respecto de la resolución de la pantalla del dispositivo en el que se visualiza la página HTML.

- Las unidades em y ex no han sido creadas por CSS, sino que llevan décadas utilizándose en el campo de la tipografía. Aunque no es una definición exacta, la unidad 1em equivale a la anchura de la letra M (*"eme mayúscula"*) del tipo y tamaño de letra del elemento.

- La unidad em hace referencia al tamaño en puntos de la letra que se está utilizando. Si se utiliza una tipografía de 12 puntos, 1em equivale a 12 puntos. El valor de 1ex se puede aproximar por 0.5 em.

3.1.3. Porcentajes

El porcentaje también es una unidad de medida relativa, aunque por su importancia CSS la trata de forma separada a em, ex y px. Un porcentaje está formado por un valor numérico seguido del símbolo % y siempre está referenciado a otra medida.

Cada una de las propiedades de CSS que permiten indicar como valor un porcentaje, define el valor al que hace referencia ese porcentaje.

Los porcentajes se pueden utilizar por ejemplo para establecer el valor del tamaño de letra de los elementos:

```
body { font-size: 1em; }
h1 { font-size: 200%; }
h2 { font-size: 150%; }
```

Los tamaños establecidos para los elementos <h1> y <h2> mediante las reglas anteriores, son equivalentes a 2em y 1.5em respectivamente, por lo que es más habitual definirlos mediante em.

3.1.4. Recomendaciones

En general, se recomienda el uso de unidades relativas siempre que sea posible, ya que mejora la accesibilidad de la página y permite que los documentos se adapten fácilmente a cualquier medio y dispositivo.

El documento Recomendaciones sobre técnicas CSS para la mejora de la accesibilidad de los contenidos HTML, elaborado por el organismo W3C, recomienda el uso de la unidad em para indicar el tamaño del texto y para todas las medidas que sean posibles.

Normalmente se utilizan píxel y porcentajes para definir el layout del documento (básicamente, la anchura de las columnas y de los elementos de las páginas) y em y porcentajes para el tamaño de letra de los textos.

unidad em

3.2. Colores

3.2.1. Palabras clave

CSS define 17 palabras clave para referirse a los colores básicos. Las palabras se corresponden con el nombre en inglés de cada color:

Figura 3.1 Colores definidos mediante las palabras clave de CSS

3.2.2. RGB decimal

En el campo del diseño gráfico, se han definido varios modelos para hacer referencia a los colores. Los dos modelos más conocidos son RGB y CMYK. Simplificando su explicación, el modelo RGB consiste en definir un color indicando la cantidad de color rojo, verde y azul que se debe *mezclar* para obtener ese color. Técnicamente, el modelo RGB es un modelo de tipo "aditivo", ya que los colores se obtienen sumando sus componentes.

```
p { color: rgb(71, 98, 176); }
```

3.2.3. RGB porcentual

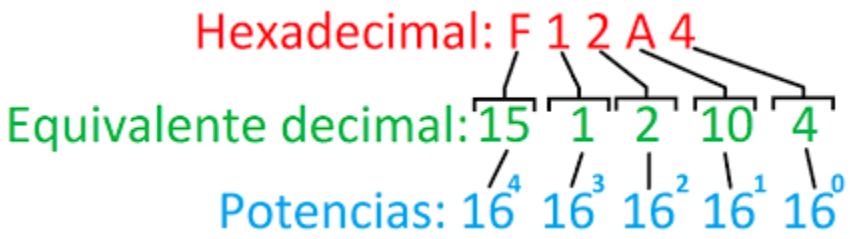
Las componentes RGB de un color también se pueden indicar mediante un porcentaje. El funcionamiento y la sintaxis de este método es el mismo que el del RGB decimal.

```
p { color: rgb(27%, 38%, 69%); }
```

3.2.4. RGB hexadecimal

Aunque es el método más complicado para indicar los colores, se trata del método más utilizado con mucha diferencia. De hecho, prácticamente todos los sitios web reales utilizan exclusivamente este método.

Para entender el modelo RGB hexadecimal, en primer lugar, es preciso introducir un concepto matemático llamado sistema numérico hexadecimal. Cuando realizamos operaciones matemáticas, siempre utilizamos 10 símbolos para representar los números (del 0 al 9). Por este motivo, se dice que utilizamos un sistema numérico decimal.



¿Cuál es el valor en decimal de FA?

Decimal	Binario	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Definir un color en CSS con el método RGB hexadecimal requiere realizar los siguientes pasos:

- Determinar las componentes RGB decimales del color original, por ejemplo: R = 71, G = 98, B = 176
- Transformar el valor decimal de cada componente al sistema numérico hexadecimal.
- Se trata de una operación exclusivamente matemática, por lo que puedes utilizar una calculadora.
- En el ejemplo anterior, el valor hexadecimal de cada componente es: R = 47, G = 62, B = B0
- Para obtener el color completo en formato RGB hexadecimal, se concatenan los valores hexadecimales de las componentes RGB en ese orden y se les añade el prefijo #.
- De esta forma, el color del ejemplo anterior es #4762B0 en formato RGB hexadecimal.

Siguiendo el mismo ejemplo de las secciones anteriores, el color del párrafo se indica de la siguiente forma utilizando el formato RGB hexadecimal:

```
p { color: #4762B0; }
```

Capítulo 4. Modelo de cajas

El modelo de cajas o *"box model"* es seguramente la característica más importante del lenguaje de hojas de estilos CSS, ya que condiciona el diseño de todas las páginas web. El modelo de cajas es el comportamiento de CSS que hace que todos los elementos de las páginas se representen mediante cajas rectangulares.

Las cajas de una página se crean automáticamente. Cada vez que se inserta una etiqueta HTML, se crea una nueva caja rectangular que encierra los contenidos de ese elemento. La siguiente imagen muestra las tres cajas rectangulares que crean las tres etiquetas HTML que incluye la página:

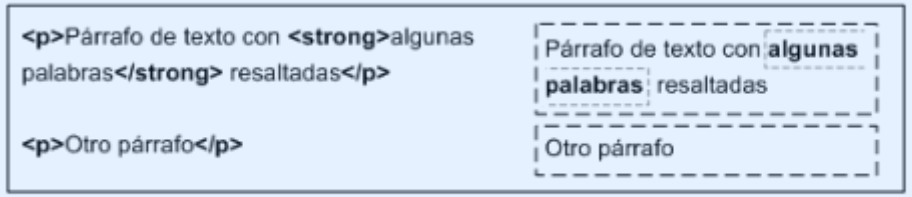


Figura 4.1 Las cajas se crean automáticamente al definir cada elemento HTML

Los navegadores crean y colocan las cajas de forma automática, pero CSS permite modificar todas sus características. Cada una de las cajas está formada por seis partes, tal y como muestra la siguiente imagen:

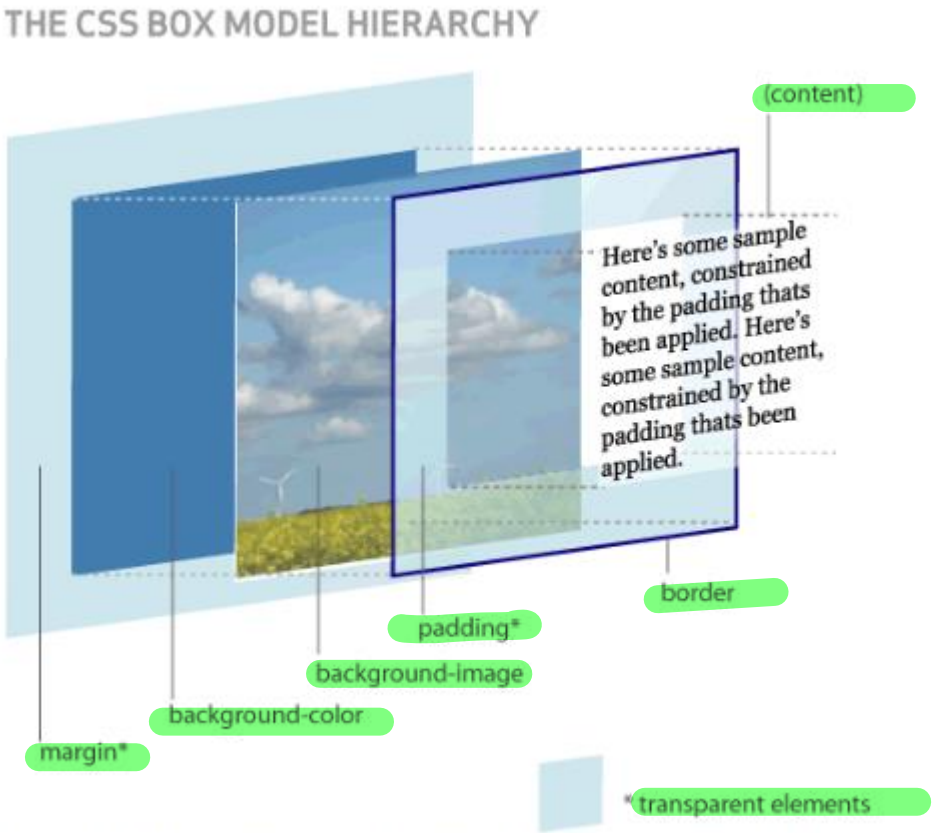


Figura 4.3 Representación tridimensional del box model de CSS

Las partes que componen cada caja y su orden de visualización desde el punto de vista del usuario son las siguientes:

- ***Contenido (content)***: se trata del contenido HTML del elemento (las palabras de un párrafo, una imagen, el texto de una lista de elementos, etc.)
- ***Relleno (padding)***: espacio libre opcional existente entre el contenido y el borde.
- ***Borde (border)***: línea que encierra completamente el contenido y su relleno.
- ***Imagen de fondo (background image)***: imagen que se muestra por detrás del contenido y el espacio de relleno.
- ***Color de fondo (background color)***: color que se muestra por detrás del contenido y el espacio de relleno.
- ***Margen (margin)***: separación opcional existente entre la caja y el resto de las cajas adyacentes.

4.1. Anchura y altura

La propiedad CSS que controla la anchura de la caja de los elementos se denomina width.

Propiedad	width
Valores	unidad de medida | porcentaje | auto | inherit
Se aplica a	Todos los elementos, salvo los elementos en línea que no sean imágenes, las filas de tabla y los grupos de filas de tabla
Valor inicial	auto
Descripción	Establece la anchura de un elemento

Propiedad	height
Valores	unidad de medida | porcentaje | auto | inherit
Se aplica a	Todos los elementos, salvo los elementos en línea que no sean imágenes, las columnas de tabla y los grupos de columnas de tabla
Valor inicial	auto
Descripción	Establece la altura de un elemento

4.2. Margen y relleno

Propiedades	margin-top, margin-right, margin-bottom, margin-left
Valores	unidad de medida | porcentaje | auto | inherit
Se aplica a	Todos los elementos, salvo margin-top y margin-bottom que sólo se aplican a los elementos de bloque y a las imágenes
Valor inicial	0
Descripción	Establece cada uno de los márgenes horizontales y verticales de un elemento

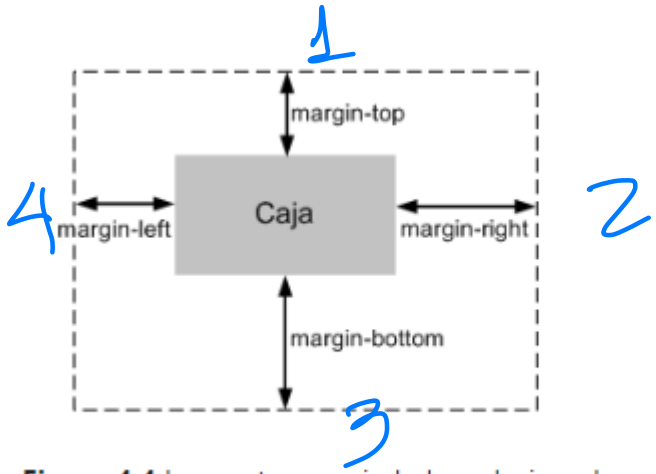


Figura 4.4 Las cuatro propiedades relacionadas con los márgenes

Código CSS original:

```
div img {
        margin-top: .5em;
        margin-bottom: .5em;
        margin-left: 1em;
        margin-right: .5em; }
```

Alternativa directa:

```
div img { margin: .5em .5em .5m 1em; }
```

4.2.2. Relleno

Propiedades	padding-top, padding-right, padding-bottom, padding-left
Valores	unidad de medida | porcentaje | inherit
Se aplica a	Todos los elementos excepto algunos elementos de tablas como grupos de cabeceras y grupos de pies de tabla
Valor inicial	0
Descripción	Establece cada uno de los rellenos horizontales y verticales de un elemento

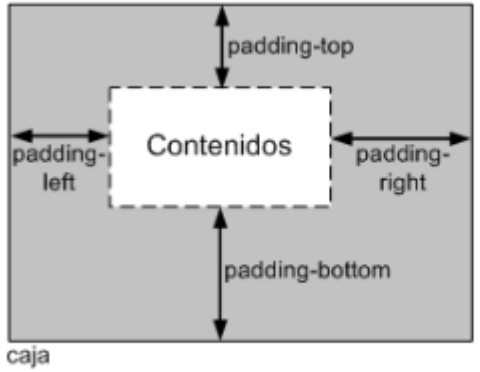


Figura 4.10 Las cuatro propiedades relacionadas con los rellenos

```
body {padding: 2em} /* Todos los rellenos valen 2em */

body {padding: 1em 2em} /* Superior e inferior = 1em, Izquierdo y derecho = 2em */

body {padding: 1em 2em 3em} /* Superior = 1em, derecho = 2em, inferior = 3em, izquierdo = 2em */

body {padding: 1em 2em 3em 4em} /* Superior = 1em, derecho = 2em, inferior = 3em, izquierdo = 4em */
```

4.3. Bordes

4.3.1. Anchura

La anchura de los bordes se controla con las cuatro propiedades siguientes:

Propiedades	border-top-width, border-right-width, border-bottom-width, border-left-width
Valores	(unidad de medida | thin | medium | thick) | inherit
Se aplica a	Todos los elementos
Valor inicial	Medium
Descripción	Establece la anchura de cada uno de los cuatro bordes de los elementos

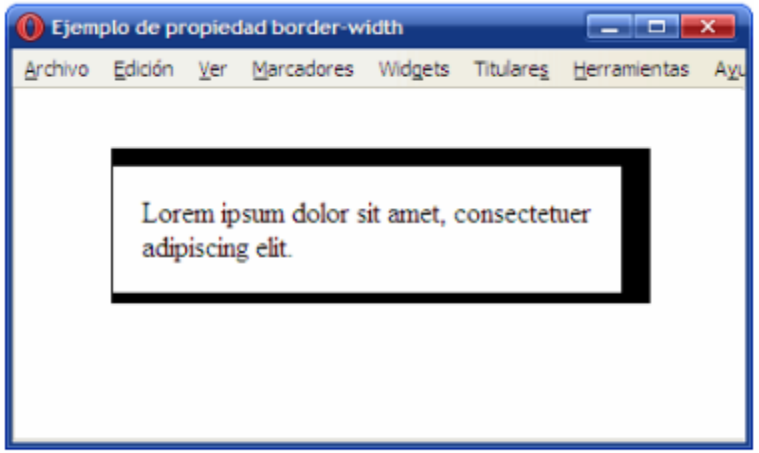


Figura 4.11 Ejemplo de propiedad border-width

Las reglas CSS utilizadas se muestran a continuación:

```
div {
        border-top-width: 10px;
        border-right-width: 1em;
        border-bottom-width: thick;
        border-left-width: thin;
}
```

Si se quiere establecer de forma simultánea la anchura de todos los bordes de una caja, es necesario utilizar una propiedad "shorthand" llamada border-width:

Propiedad	border-width
Valores	(unidad de medida | thin | medium | thick) {1, 4} | inherit
Se aplica a	Todos los elementos
Valor inicial	Medium
Descripción	Establece la anchura de todos los bordes del elemento

4.3.2. Color

El color de los bordes se controla con las cuatro propiedades siguientes:

Propiedades	border-top-color, border-right-color, border-bottom-color, border-left-color
Valores	color | transparent | inherit
Se aplica a	Todos los elementos
Valor inicial	-
Descripción	Establece el color de cada uno de los cuatro bordes de los elementos

```
div {
border-top-color: #CC0000;
border-right-color: blue;
border-bottom-color: #00FF00;
border-left-color: #CCC;
}
```

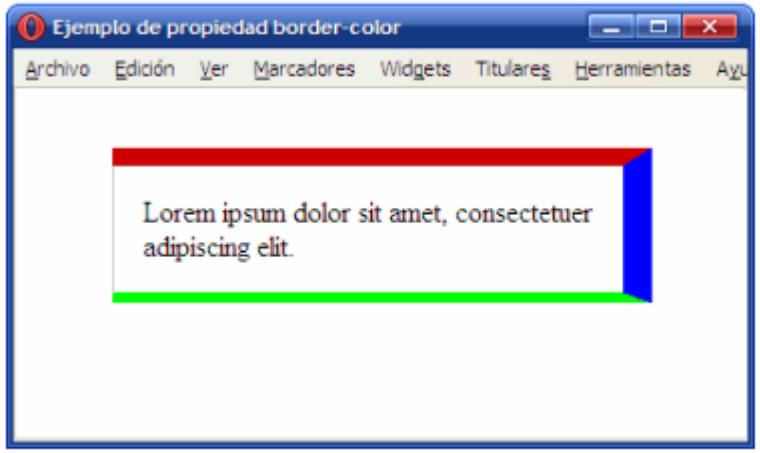


Figura 4.12 Ejemplo de propiedad border-color

4.4. Margen, relleno, bordes y modelo de cajas

- La anchura y altura de un elemento no solamente se calculan teniendo en cuenta sus propiedades width y height.

- El margen, el relleno y los bordes establecidos a un elemento determinan la anchura y altura final del elemento. En el siguiente ejemplo se muestran los estilos CSS de un elemento:

```
div {
        width: 300px;
        padding-left: 50px;
        padding-right: 50px;
        margin-left: 30px;
        margin-right: 30px;
        border: 10px solid black;
}
```

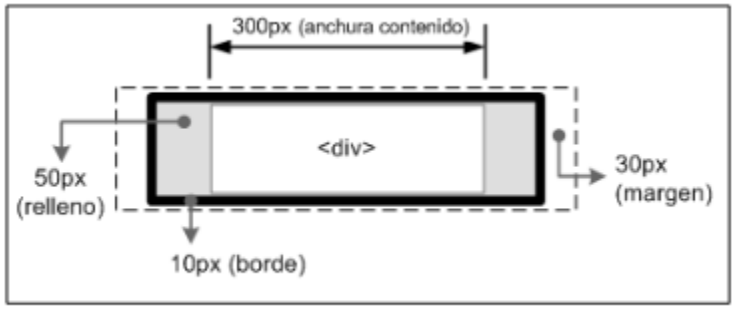


Figura 4.15 La anchura total de un elemento tiene en cuenta los márgenes, rellenos y bordes

4.5. Fondos

El último elemento que forma el *box model* es el fondo de la caja del elemento. El fondo puede ser un color simple o una imagen. El fondo solamente se visualiza en el área ocupada por el contenido y su relleno, ya que el color de los bordes se controla directamente desde los bordes y las zonas de los márgenes siempre son transparentes.

Para establecer un color o imagen de fondo en la página entera, se debe establecer un fondo al elemento <body>. Si se establece un fondo a la página, como el valor inicial del fondo de los elementos es transparente, todos los elementos de la página se visualizan con el mismo fondo a menos que algún elemento especifique su propio fondo.

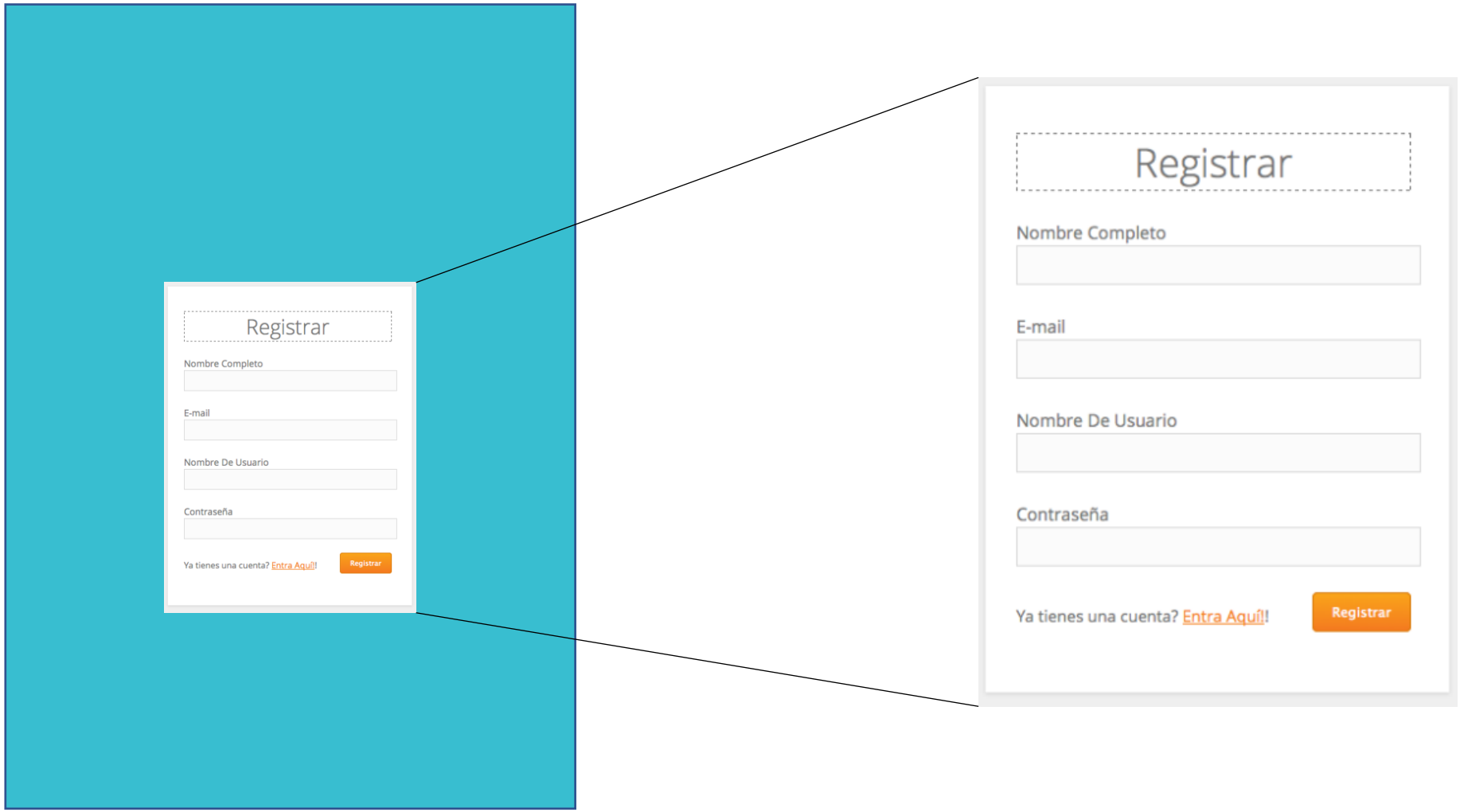
Propiedad	background-color
Valores	color | transparent | inherit
Se aplica a	Todos los elementos
Valor inicial	transparent
Descripción	Establece un color de fondo para los elementos

```
body { background-color: #F5F5F5; }
```

Propiedad	background-image
Valores	url | none | inherit
Se aplica a	Todos los elementos
Valor inicial	none
Descripción	Establece una imagen como fondo para los elementos

```
body { background-image: url("imagenes/fondo.png") }
```

Realizar un formulario como el de la siguiente figura aplicando estilos css



INTRODUCCIÓN
A JAVASCRIPT
JS

Sintaxis (I): Inclusión de código y generalidades

Lo primero que uno debe saber antes de programar nada en JavaScript, es cómo incrustarlo en un documento HTML. Para ello, tenemos dos formas; una, usando una directiva especial y otra, como parámetro de ciertas directivas (como <A>, <IMG>,...) en respuesta a eventos de usuario ya predefinidos (pulsar el ratón, cargarse la página,...).

```
<SCRIPT LANGUAGE="JavaScript">
<!--
Código JavaScript
//-->
</SCRIPT>
```

¿Y dónde colocamos este código?
En principio no importa mucho dónde lo pongamos, sin embargo, una **buena costumbre** es introducirlo en la **cabecera** del documento HTML (entre <HEAD> . . . </HEAD>), puesto que así estamos completamente seguros de que cuando se empieza a cargar el documento y a aparecer en la pantalla, todo el código JavaScript ya está cargado y listo para ser ejecutado si se da el evento apropiado.

Generalidades

- Es "Case Sensitive", es decir, distingue mayúsculas de minúsculas.
- Comentarios: `/* ... */` para encerrar un bloque y `//` para comentarios de una línea.
- Cada sentencia ha de terminar en `;`
- Encerrando código entre llaves `{ ... }` lo agrupamos. Se verá su sentido cuando tratemos las estructuras de control.

Sintaxis (I): Variables y constantes

1. Variables
2. Ámbito de las variables
3. Constantes

JavaScript tiene la peculiaridad de ser un lenguaje ***débilmente tipado***, esto es, se puede declarar una variable que ahora sea un entero y más adelante una cadena.

Para declarar variables no tenemos más que poner la palabra `var` y a continuación la lista de variables separadas por comas. No todos los nombres de variable son válidos, hay unas pocas restricciones:

- Un nombre válido de variable no puede tener espacios.
- Puede estar formada por números, letras y el caracter subrayado `_`
- No se puede usar palabras reservadas (`if`, `for`, `while`, `break`...).
- No pueden empezar por un número, es decir, el primer caracter del nombre de la variable ha de ser una letra o `_`

Ámbito de las variables

En JS también tenemos variables locales y variables globales. Las variables **locales** serán aquellas que se definan dentro de una función, mientras que las variables **globales** serán aquellas que se definan fuera de la función, y podrán ser consultadas y modificadas por cualquiera de las funciones que tengamos en el documento HTML. Un buen sitio para definir variables globales es en la cabecera, `<HEAD> ... </HEAD>`

Constantes

Las constantes no existen como tales en JavaScript. Salvando el caso del objeto `Math`, que veremos más adelante, no las tenemos disponibles. Si se quiere usar una constante en el programa, no quedará más remedio que declarar una variable, asignarle un valor desde el principio del programa, y procurar no tocarlo. Es lo más parecido a las constantes que se puede obtener.

Sintaxis (I): Tipos de datos

1. Enteros
2. Flotantes
3. Booleanos
4. Nulos
5. Indefinidos
6. Cadenas
7. Objetos
8. Funciones

El tipo entero y el tipo flotante se engloban dentro del tipo `Number`. Los `nulos` y los `indefinidos` no son realmente tipos de datos: son valores especiales. Es más, para JavaScript, el tipo nulo (`null`) es un objeto.

Enteros

Podemos representar números enteros de tres formas distintas: en base 10 (la usual), en base 16 (hexadecimal) o en base 8 (octal). Para denotar un número hexadecimal lo haremos escribiendo delante del número **0x**. Por ejemplo, `0xF3A2` se refiere a `F3A2` en hexadecimal. Para denotar un número en base octal lo haremos precediéndolo de un **0** (cero). Por ejemplo, `01763` se refiere a `1763` en octal.

Flotantes

Con ellos podremos representar números en coma flotante, usando notación decimal o científica.

Por ejemplo, el número `123.298` se escribe `1.23298E2`, mientras que `0.00123` se escribe `1.23E-3`

Booleanos

Tomarán los valores de verdad `true` o `false` (1 ó 0). El resultado de evaluar una condición será un valor booleano.

Nulos

`null` es un valor especial que nos dice que un objeto no ha sido asignado.

Indefinidos

El valor `undefined` (indefinido) nos dice que la variable a la que queremos acceder no le ha sido asignado valor alguno, y por tanto permanece indefinida.

Cadenas

En JavaScript, las cadenas vienen delimitadas o bien por comillas dobles, o bien por comillas simples, y pueden tener cualquier combinación de letras, espacios, números y otros símbolos.

¿A qué se debe que sea válido tanto encerrar las cadenas entre comillas dobles como encerrarlas entre comillas simples?

Muchas veces, habrá alguna función JS incrustada en alguna directiva HTML, y esta usa las comillas dobles; en este caso, usaremos las comillas simples para no cerrar el valor del parámetro que estábamos dando a la directiva HTML.

Podemos, además, usar los caracteres de escape que tenemos en C para representar los saltos de línea, tabulaciones, etc...

```
\b   Espacio hacia atrás
\f   Alimentación de línea
\n   Nueva línea
\r   Retorno de carro
\t   Tabulación
\\   Barra invertida: \
\'   Comilla simple: '
\"   Comilla doble: "
```

Objetos

Una variable puede ser un objeto, en cuyo caso corresponde a este tipo. Veremos con detalle los objetos en un tema aparte.

Funciones

Las funciones también son un tipo de dato especial: si una variable es de tipo `function`, en realidad se trata de una función.

Operadores

Operadores: Aritméticos

```
+      Suma
-      Resta
*      Producto
/      Cociente
%      Módulo
```

Operadores: Comparación

```
==        igual
!=        distinto
>         mayor que
<         menor que
>=        mayor o igual que
<=        menor o igual que
```

```
===        estrictamente igual
!==        estrictamente distinto
```

El primero devuelve **`true`** cuando los operadores toman el mismo valor y además son del mismo tipo. El segundo devuelve **`true`** cuando los operadores son distintos y de tipo distinto.

Operadores: Lógicos

```
&&     AND ('y' lógica)
||     OR ('o' lógica)
!      NOT ('no' lógica)
```

Operadores: A nivel de bit

```
&        AND bit a bit
|        OR  bit a bit
^        XOR bit a bit
~        NOT bit a bit
>>       rotación a derecha
<<       rotación a izquierda
```

Operadores: Resto

```
new, delete, void, typeof
```

`new` es un operador que sirve para crear nuevas instancias de un objeto vía su constructor.

`delete` es un operador que sirve para destruir una instancia concreta de un objeto.

`void` es un operador que permite evaluar una expresión sin que devuelva valor alguno. Podría considerarse como una función especial.

`typeof` es un operador que nos devuelve una cadena que describe el tipo de dato que corresponde con el objeto (variable, función, etc)

Sintaxis (II): Estructuras de control. Funciones

Sintaxis (II): Condicionales

El condicional if

```
if(condición) {
codigo necesario }
else {
codigo alternativo }
```

```
[ Expresión ] ? [ Sentencia1 ] : [ Sentencia2 ] ;
```
//condicional ternario

que evalúa `[ Expresión ]`. El resultado es un booleano. Si es `'true'` se ejecutará `[ Sentencia1 ]` y si es `'false'`, se ejecutará `[ Sentencia2 ]`

El condicional switch

```
switch(condicion) {
  case caso1 :
    sentencias para caso1;
    break;
       ......
  case casoN :
    sentencias para casoN;
    break;
  default :
    sentencias por defecto;
    break;
  }
```

Sintaxis (II): Bucles

1. El bucle `for`
2. El bucle `while`
3. El bucle `do ... while`
4. El bucle `for ... in`
5. La construcción `with`

El bucle `for`

```
for([Inicialización]; [Condición]; [Expresión de actualización])
{
Instrucciones a repetir
}
```

El bucle `while`

```
while(Condición) {
 Instrucciones a repetir
 }
```

El bucle `do ... while`

```
do {
 Instrucciones a repetir
 } while(condicion);
```

El bucle `for ... in`

```
for(var 'Propiedad' in 'Objeto') {
  Instrucciones a repetir
  }
```

Sintaxis (II): Funciones

```
function NombreFuncion(arg1, ..., argN)  {
  Código de la función

  return Valor;
  }
```

Las funciones en JavaScript tienen una propiedad particular, y es que `no tienen un número fijo de argumentos`. Es decir, nosotros podremos llamar a las funciones con un número cualquiera de argumentos y con cualquier tipo de argumentos.

Los argumentos pueden ser accedidos bien por su nombre, bien por un vector de argumentos que tiene asociada la función (que será `NombreFuncion.arguments`), y podemos saber cuántos argumentos se han entrado viendo el valor de `NombreFuncion.arguments.length`

```
<script language="JavaScript">
 function Cuenta() {
 document.write("Me han llamado con " + Cuenta.arguments.length  + " argumentos<P>");
 for(i = 0; i < Cuenta.arguments.length; i++)
 document.write("Argumento " + i + ": " + Cuenta.arguments[i] + "<BR>");
 }

 <!--
 Cuenta("Hola","a","todos");
 //-->
</script>
```

Introducción a los objetos.

```
OBJETO = DATOS + FUNCIONES QUE TRABAJAN CON ESOS DATOS
```

```
CLASE Ventana
 VARIABLES
  x0, y0, x1, y1: ENTEROS
 FUNCIONES
  Inicializar_Coordenadas()
  Dibujar_Ventana()
  Mover_Ventana()
FIN CLASE Ventana
```

```
Ventana Ventana1, Ventana2
```

A la hora de acceder a las variables que tiene cada una de estas ventanas (cuyos valores son distintos para cada ventana) o de utilizar las funciones de cada una de estas ventanas, habría que poner:

```
Ventana1.x0 = 3
Ventana2.Dibujar_Ventana()
```

es decir:

```
Objeto.Variable
Objeto.Funcion(argumentos)
```

Cuando se habla de POO se habla también de acceso a las partes de un objeto (partes públicas, privadas y protegidas), pero como JS es más simple, no tenemos distinción de accesos.

Sin embargo, la forma de inicializar las coordenadas de la ventana más correcta no sería haciendo `Ventana1.x0=3;` , etc., como he puesto en el ejemplo, sino que se encargará de ello una de sus propias funciones (una función especial llamada 'constructor'), pasándole como argumentos los valores de inicialización.

En JS, el constructor es precisamente la función con la que crearemos el molde de la clase.

Objetos: Cómo son en JavaScript

```
function Mi_Objeto(dato_1, ..., dato_N)  {
  this.variable_1 = dato_1;
            ....
  this.variable_N = dato_N;

  this.funcion_1 = Funcion_1;
           ....
  this.funcion_N = Funcion_N;
  }
```

Los objetos son una colección de propiedades
Para construir objetos podemos hacerlo de dos maneras,

•**Objetos declarativos o literales**: podemos crear objetos sin necesidad de un constructor o instanciar una clase, para esto solo declaramos el objeto y sus propiedades.

```
const camilo = {
nombre: 'Camilo’,
edad: 22,
sexo: 'masculino’,
pasatiempos: ['patinar’,'bailar’],
hablar: function(){ return `hola soy ${this.nombre}, y tengo ${this.edad} años`; } }
console.log(camilo);
```

•**Objetos construidos:** JavaScript es un lenguaje libre de clases, pero tenemos el keyword *new*, el cual nos permite crear un nuevo objeto, de esta manera podemos utilizar una función que cumpla el rol del constructor.

```
function Persona(nombre, edad, sexo, pasatiempos) {
this.nombre = nombre;
this.edad = edad;
this.sexo = sexo;
this.pasatiempos = pasatiempos;
this.hablar = function() { return `hola soy ${this.nombre}, y tengo ${this.edad} años`; }; }

const camilo = new Persona('camilo', 22, 'masculino', ['patinar', 'bailar']);

console.log(camilo)
```

Contenido

JavaScript no almacena el contenido de las propiedades dentro de los objetos, este solo guarda el nombre de las propiedades, con referencias a donde están almacenados los valores.

```
const myObj = {
nombre: 'yeison’,
hablar: function(){ return `hola soy ${this.nombre}`; } }

const myFunc = myObj.hablar;
myObj.hablar = null;

console.log(myObj.hablar); //null

console.log(myFunc); // function hablar() {..}
```

Información adicional TypScript

Instalación de Node

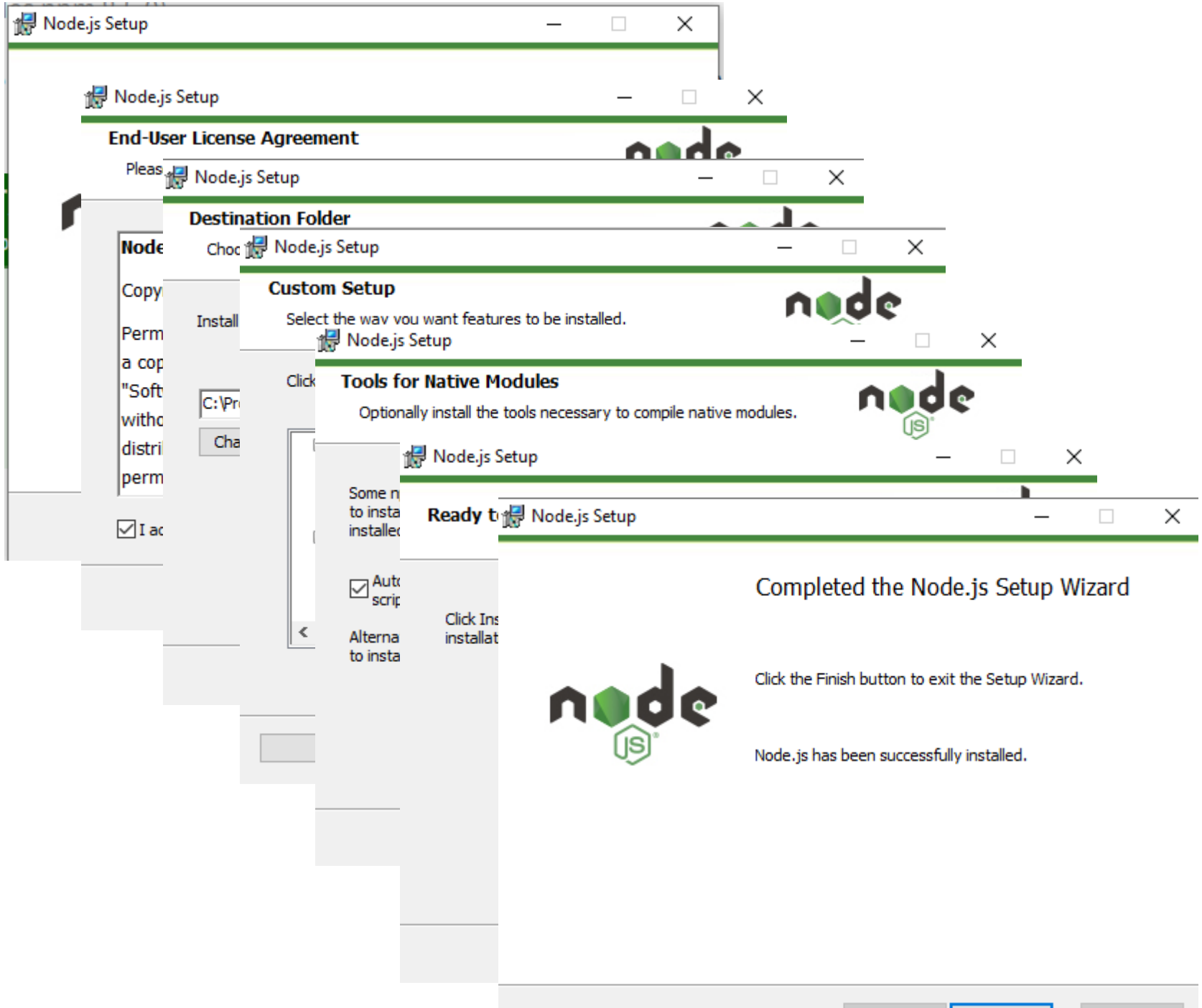
Descargas

Versión actual: **16.14.2** (includes npm 8.5.0)

Descargue el código fuente de Node.js o un instalador pre-compilado para su plataforma, y comience a desarrollar hoy.

LTS Recomendado para la mayoría	Actual Últimas características	
Instalador Windows node-v16.14.2-x64.msi	Instalador macOS node-v16.14.2.pkg	Código Fuente node-v16.14.2.tar.gz

Instalador Windows (.msi)	32-bit	64-bit
Binario Windows (.zip)	32-bit	64-bit
Instalador macOS (.pkg)	64-bit / ARM64	
Binario macOS (.tar.gz)	64-bit	ARM64
Binario Linux (x64)	64-bit	
Binario Linux (ARM)	ARMv7	ARMv8
Código Fuente	node-v16.14.2.tar.gz	


Node.js Setup
End-User License Agreement
Destination Folder
Custom Setup
Select the way you want features to be installed.
Tools for Native Modules
Optionally install the tools necessary to compile native modules.
Completed the Node.js Setup Wizard
Click the Finish button to exit the Setup Wizard.
Node.js has been successfully installed.

Símbolo del sistema

```
Microsoft Windows [Versión 10.0.19044.1586]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\L03037251>npm install typescript --save-dev
npm WARN old lockfile
npm WARN old lockfile The package-lock.json file was created with an old version of npm,
npm WARN old lockfile so supplemental metadata must be fetched from the registry.
npm WARN old lockfile
npm WARN old lockfile This is a one-time fix-up, please be patient...
npm WARN old lockfile

added 1 package, removed 1 package, and audited 2 packages in 25s

found 0 vulnerabilities
npm notice
npm notice New patch version of npm available! 8.5.0 -> 8.5.5
npm notice Changelog: https://github.com/npm/cli/releases/tag/v8.5.5
npm notice Run npm install -g npm@8.5.5 to update!
npm notice

C:\Users\L03037251>npm install -g npm@8.5.5

removed 164 packages, changed 3 packages, and audited 31 packages in 8s

found 0 vulnerabilities

C:\Users\L03037251>npm install -g @angular/cli

added 14 packages, removed 57 packages, changed 177 packages, and audited 192 packages in 23s

23 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\Users\L03037251>
```

TypeScript

TypeScript es un lenguaje de programación orientado a objetos fuertemente tipado que se traduce a ***JavaScript*** añadiéndole características que no posee. La operación de traducir TypeScript a JavaScript se conoce como ***transpilación***.

Una vez escritos, dichos programas se transpilan (compilan) o, lo que es igual, se traducen a JavaScript para que puedan ejecutarse posteriormente. Cualquier instrucción escrita en JavaScript “puro” se copia igual (sin traducción) en el fichero resultante de la transpilación.

Para compilar usaremos el comando ***TSC*** y para ejecutar emplearemos el comando ***node***, como veremos más adelante y el cual ya hemos instalado en capítulos anteriores.

La sintaxis que hallaremos en ***TypeScript*** está compuesta por módulos, funciones, variables, sentencias, expresiones y comentarios.

Respecto de las funciones, ***TypeScript*** permite definir funciones con nombre, anónimas, tipo Lambda o de flecha, además de permitir la sobrecarga de funciones utilizando diferentes parámetros y/o tipos en sus llamadas.

Tipo operador	Operadores
Aritméticos	+, -, *, /, %, ++, --
Relacionales	>, <, >=, <=, ==, !=
Lógicos	&&, ||, !
Asignaciones	=, +=, -=, *=, /=
Condicional	test ? expre1 : expre2
Typeof	Devuelve tipo de datos del operando

```
function nombre_funcion():tipo_retorno
{
    //sentencias
return valor;
```

Numéricos	String	
• toExponential()	• charAt()	• slice()
• toFixed()	• charCodeAt()	• split()
• toLocaleString()	• concat()	• substr()
• toPrecision()	• indexOf()	• substring()
• toString()	• lastIndexOf()	• toLocaleLowerCase()
• valueOf()	• localeCompare()	• toLocaleUpperCase()
	• match()	• toLowerCase()
	• replace()	• toString()
	• search()	• toUpperCase()
		• valueOf()

Actividad 1

A continuación, vamos a crear un programa con TypeScript que simplemente muestre el texto **Hola Mundo**:

1. Para ello, nos ubicaremos en nuestro directorio de ejercicios (**Ej_angular**) y crearemos el directorio **HolaMundo_TS**. (los ficheros deben tener la extensión .ts)

2. Seguidamente, abriremos nuestro editor favorito y, dentro del directorio recién creado, crearemos el fichero **HMTS.ts** con el siguiente contenido

```
var texto:string=”Hola Mundo”
console.log(texto)
```

3. Una vez creado y guardado el fichero, accederemos a la ventana de **CMD** y nos ubicaremos en nuestro nuevo directorio (**HolaMundo_TS**) para teclear lo siguiente:

```
tsc HMTS.ts
```

4. En este punto, procederemos a la ejecución de nuestro programa tecleando lo siguiente:

```
node HMTS
```

Tipos básicos de Datos

El tipo de datos Null

Al igual que en JavaScript, el tipo de datos `null` en TypeScript solo puede tener un valor válido: `null`. Una variable null no puede contener otros tipos de datos como número y cadena de texto. Establecer una variable a null borrará su contenido si tuviese alguno.

```
01  // With strictNullChecks set to true
02  let a: null = null; // Ok
03  let b: undefined = null; // Error
04  let c: number = null; // Error
05  let d: void = null; // Error
06
07  // With strictNullChecks set to false
08  let a: null = null; // Ok
09  let b: undefined = null; // Ok
10  let c: number = null; // Ok
11  let d: void = null; // Ok
```

El Tipo de Datos Void

El tipo de datos void se usa para indicar la falta de un `type` para una variable. Establecer variables para que tengan un tipo `void` puede no ser muy útil, pero usted puede establecer el tipo de retorno de las funciones que no devuelven nada a `void`. Cuando se usa con variables, el tipo `void` solo puede tener dos valores válidos: `null` y `undefined`.

```
01  // With strictNullChecks set to true
02  let a: void = undefined; // Ok
03  let b: void = null; // Error
04  let c: void = 3; // Error
05  let d: void = "apple"; // Error
06
07  // With strictNullChecks set to false
08  let a: void = undefined; // Ok
09  let b: void = null; // Ok
10  let c: void = 3; // Error
11  let d: void = "apple";  // Error
```

El Tipo de Datos Boolean

A diferencia de los tipos de datos `number` y `string`, `boolean` solo tiene dos valores válidos.

```
1  let a: boolean = true;
2  let b: boolean = false;
3  let c: boolean = 23; // Error
4  let d: boolean = "blue"; // Error
```

El Tipo de Datos Numérico

El tipo de datos `number` se usa para representar tanto enteros como valores de punto flotante en JavaScript y en TypeScript. Sin embargo, debe recordar que todos los números están representados internamente como valores de coma flotante. Los números también se pueden especificar como literales Hexadecimales, Octales o Binarios. Tenga en cuenta que las representaciones Octal y Binaria se introdujeron en ES6, y esto puede dar como resultado una salida de código JavaScript diferente en función de la versión a la que está apuntando.

```
01  // With strictNullChecks set to true
02  let a: number = undefined; // Error
03  let b: number = null; // Error
04  let c: number = 3;
05  let d: number = 0b111001; // Binary
06  let e: number = 0o436; // Octal
07  let f: number = 0xadf0d; // Hexadecimal
08  let g: number = "cat"; // Error
09
10  // With strictNullChecks set to false
11  let a: number = undefined; // Ok
12  let b: number = null; // Ok
13  let c: number = 3;
14  let d: number = 0b111001; // Binary
15  let e: number = 0o436; // Octal
16  let f: number = 0xadf0d; // Hexadecimal
17  let g: number = "cat"; // Error
```

El Tipo de Datos String

El tipo de datos string se utiliza para almacenar información textual. Tanto JavaScript como TypeScript usan comillas dobles ("), así como comillas simples (') para rodear su información textual como una cadena. Una cadena puede contener cero o más caracteres entre comillas.

```
01  // With strictNullChecks set to true
02  let a: string = undefined; // Error
03  let b: string = null; // Error
04  let c: string = "";
05  let d: string = "y";
06  let e: string = "building";
07  let f: string = 3; // Error
08  let g: string = "3";
09
10  // With strictNullChecks set to false
11  let a: string = undefined; // Ok
12  let b: string = null; // Ok
13  let c: string = "";
14  let d: string = "y";
15  let e: string = "building";
16  let f: string = 3; // Error
17  let g: string = "3";
```

El Tipo de Datos Indefinido

Cualquier variable cuyo valor no haya especificado se establece en `undefined`. Sin embargo, usted también puede establecer explícitamente el tipo de una variable como indefinida, como se ve en el siguiente ejemplo.

una variable con `type` configurado como `undefined` solo puede tener undefined como su valor.

```
01  // With strictNullChecks set to true
02  let a: undefined = undefined; // Ok
03  let b: undefined = null; // Error
04  let c: number = undefined; // Error
05  let d: void = undefined; // Ok
06
07  // With strictNullChecks set to false
08  let a: undefined = undefined; // Ok
09  let b: undefined = null; // Ok
10  let c: number = undefined; // Ok
11  let d: void = undefined; // Ok
```

Los tipos de datos Array y Tuple

Puede definir tipos de array de dos maneras diferentes en JavaScript. En el primer método, usted especifica el tipo de elementos de array seguidos por `[]` lo cual que denota un array de ese tipo. Otro método es utilizar el tipo de array genérico `Array<elemType>` . El siguiente ejemplo muestra cómo crear arrays con ambos métodos. Especificar `null` o `undefined` como uno de los elementos producirá errores cuando el indicador `strictNullChecks` sea `true`.

```
01  // With strictNullChecks set to false
02  let a: number[] = [1, 12, 93, 5];
03  let b: string[] = ["a", "apricot", "mango"];
04  let c: number[] = [1, "apple", "potato"]; // Error
05
06  let d: Array<number> = [null, undefined, 10, 15];
07  let e: Array<string> = ["pie", null, ""];
08
09
10  // With strictNullChecks set to true
11  let a: number[] = [1, 12, 93, 5];
12  let b: string[] = ["a", "apricot", "mango"];
13  let c: number[] = [1, "apple", "potato"]; // Error
14
15  let d: Array<number> = [null, undefined, 10, 15]; // Error
16  let e: Array<string> = ["pie", null, ""]; // Error
```

El tipo de datos de tupla le permite crear un array donde el tipo de un número fijo de elementos se conoce de antemano. El tipo del resto de los elementos solo puede ser uno de los tipos que ya ha especificado para la tupla.

```
let a: [number, string] = [11, "monday"];
let b: [number, string] = ["monday", 11]; // Error
let c: [number, string] = ["a", "monkey"]; // Error
let d: [number, string] = [105, "owl", 129, 45, "cat"];
let e: [number, string] = [13, "bat", "spiderman", 2];

e[13] = "elephant";
e[15] = false; // Error
```

El Tipo de Datos Enum

El tipo de datos `enum` está presente en muchos lenguajes de programación como C y Java. Ha faltado en JavaScript, pero TypeScript le permite crear y trabajar con enums. Si no sabe qué son `enums`, le permiten crear una colección de valores relacionados con nombres memorables.

```
enum Animals {cat, lion, dog, cow, monkey}
let c: Animals = Animals.cat;

console.log(Animals[3]); // cow
console.log(Animals.monkey); // 4
```

Por defecto, la numeración de enums comienza en 0, pero también puede establecer un valor diferente para el primero u otros miembros de forma manual.

```
enum Animals {cat = 1, lion, dog = 11, cow, monkey}
let c: Animals = Animals.cat;

console.log(Animals[3]); // undefined
console.log(Animals.monkey); // 13
```

Los tipos Any y Never

Digamos que usted está escribiendo un programa donde el valor de una variable lo determinan los usuarios o el código escrito en una biblioteca de terceros. En este caso, no podrá establecer el tipo de esa variable correctamente. La variable podría ser de cualquier tipo, como una cadena de texto, número o booleano. Este problema se puede resolver usando el tipo `any`. Esto también es útil cuando está creando arrays con elementos de tipos mixtos.

```
let a: any = "apple";
let b: any = 14;
let c: any = false;
let d: any[] = ["door", "kitchen", 13, false, null];

b = "people";
```

El tipo `never` se usa para representar valores que nunca se supone que ocurran. Por ejemplo, puede asignar `never` como el tipo de devolución de una función que nunca regresa.

```
let a: never; // Ok
let b: never = false; // Error
let c: never = null; // Error
let d: never = "monday"; // Error

function stuck(): never {
    while (true) {
    }
}
```

Interfaces

Las interfaces en ***TypeScript*** van a permitirnos dos cosas. Por un lado, crear nuevos tipos a partir de los existentes. Y por otro servir como modelo en la definición de clases que implementen la interfaz.

Por ejemplo. Supongamos que estamos desarrollando una aplicación para inventariar productos de una tienda. Podemos crear un nuevo tipo Producto mediante una interfaz:
Si queremos declarar una variable de tipo Producto:

```
interface Producto { nombre: string; precio: number; }
```

Pero además las interfaces de **TypeScript** son extensibles. Esto significa que podemos crear una interfaz más concreta a partir de una más general. Por ejemplo, podemos crear una interfaz para representar libros y otra para representar ordenadores:

```
enum TipoLibro {Narrativa, Poesía, Ensayo}
interface Libro extends Producto {
    numPaginas: number;
    titulo: string;
    autor: string;
    editorial: string;
    tipo: TipoLibro;
}
interface Ordenador extends Producto {
    marca: string;
    modelo: string;
    cpu: string;
    ram: number;
}
```

Constructores breves

Es muy habitual que cuando declaramos clases, el constructor lleva como argumentos los valores de los atributos de la clase, y en el cuerpo del constructor se realizan las asignaciones correspondientes:

```
class Cubo {
    private ancho: number;
    private largo: number;
    private profundo: number;

    constructor(a: number, l: number, p: number){
        this.ancho = a;
        this.largo = l;
        this.profundo = p;
    }

    volumen(): number{
        return this.ancho * this.largo * this.profundo
    }
}

let c = new Cubo(3, 5, 4);
console.log(c.volumen()); // arroja 60
```

Los constructores breves de ***TypeScript*** nos proporcionan una manera más breve pero equivalente de hacer lo mismo (un syntactic sugar).

```
class Cubo{

    constructor(private ancho: number, private largo: number, private profundo: number){ }
    volumen(): number{
         return this.ancho * this.largo * this.profundo
    }
}

let c = new Cubo(3, 5, 4);

console.log(c.volumen()); // arroja 60
```

onstru

Actividad 2 | Clases

Ahora en TypeScript vamos a crear la clase Rombo, la cual debe tener dos propiedades: DiagonalVertical y DiagonalHorizontal.

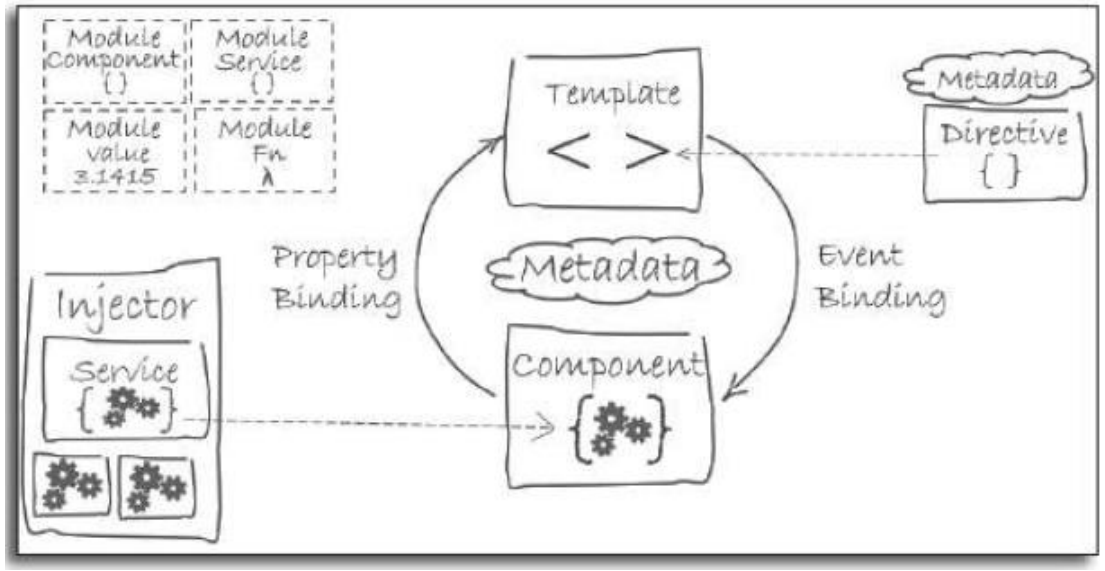
Le añadiremos un constructor breve

Y también debe de tener un método que calcule el área, que devuelve la multiplicación de DiagonalVertical * DiagonalHorizontal. Este método devolverá un número.

Información adicional

Introducción a Angular II

Componentes principales de una App en Angular



- Módulos
- Componentes
- Plantillas
- Enlace de datos
- Directivas
- Servicios
- Dependencias

¿Qué son los decoradores?

Decoradores

Básicamente es una implementación de un patrón de diseño de software que en sí sirve para extender una función mediante otra función, pero sin tocar aquella original, que se está extendiendo. El decorador recibe una función como argumento (aquella que se quiere decorar) y que devuelve esa función con alguna funcionalidad adicional.

Las funciones decoradoras comienzan por un "**@**" y a continuación tienen un nombre. Ese nombre es el de aquello que queremos decorar, que ya tiene que existir previamente. Podríamos decorar una función, una propiedad de una clase, una clase, etc.

Podemos considerar un decorador como la forma de aumentar las funcionalidades a ciertos tipos

```
1  function Bienvenida(target: Function): void{
2      target.prototype.saludo = fuction(): void{
3          console.log('¡Hola!');
4      }
5  }
6
7  @Bienvenida
8  class Saludar {
9      constructor() {
10         //implementación
11     }
12 }
13
14 var miSaludo = new Saludar();
15 miSaludo.saludo();  //Consola mostrará ¡Hola!
16
```

Definición de elementos en una aplicación

Módulo

Conjunto de códigos dedicados a resolver un objetivo que generalmente exporta una clase. Cada aplicación posee al menos una clase de módulo que denominamos “**módulo raíz**” y que, por defecto, posee el nombre de **AppModule**.

En el mismo podemos apreciar una clase con un decorador **@NgModule**, y una serie de propiedades entre las que destacamos las siguientes:

- **Declarations:** define las vistas que pertenecen a este módulo.
- **Exports:** declaraciones que han ser visibles para componentes de otros módulos.
- **Imports:** clases que otros módulos exportan para poderlas utilizar en el módulo actual.
- **Providers:** servicios usados de forma global y accesibles desde cualquier parte de la aplicación.
- **Bootstrap:** la vista principal o componente raíz, que aloja el resto de las vistas, de la aplicación. Propiedad establecida solo por el módulo raíz.

Al crear una aplicación, por defecto se crea el módulo **app.module.ts**, en el mismo, podemos apreciar diferentes partes:

Los "**imports**":

- **BrowserModule:** usado para renderizar la aplicación en el navegador.
- **NgModule:** decorador necesario para el módulo.
- **AppComponent:** componente principal del módulo.

Al decorador **@NgModule** con su metadata:

- **Declarations:** AppComponent (vista del módulo por defecto).
- **Imports:** BrowserModule (clase necesaria para el navegador).
- **Providers:** en este caso, no disponemos de ninguno.
- **Bootstrap:** AppComponent (componente principal).

A su clase: permite cargar la aplicación **(AppModule)**.

app.module.ts

src > app > app.module.ts > AppModule

```
1   import { BrowserModule } from "@angular/platform-browser";
2   import { NgModule } from "@angular/core";
3   import { BrowserAnimationsModule } from "@angular/platform-browser/animations";
4
5   import { AppComponent } from "./app.component";
6
7   @NgModule({
8     declarations: [AppComponent],
9     imports: [BrowserModule, BrowserAnimationsModule],
10    providers: [],
11    bootstrap: [AppComponent]
12  })
13  export class AppModule {}
14
```

Componente

Elemento que controla una zona de espacio de la pantalla que representa la “vista”. Define las propiedades y métodos que usa el propio template y contiene la lógica y la funcionalidad que utiliza la vista. Pueden tener atributos tanto de entrada (decorador **@Input**) como de salida (decorador **@Output**).

También podemos definir al componente como:

Componente = template + metadatos + clase

Template

Define la vista de un **Componente** mediante un fragmento de **HTML**.

Un ***componente*** es una clase a la que se le añade un decorador **(@Component)** y que permite crear nuevas etiquetas **HTML** mediante las cuales podemos añadir funcionalidad a nuestras páginas controlando determinadas zonas de pantalla. Básicamente, los componentes se organizan en forma de árbol teniendo un componente principal donde, por defecto, su propiedad "selector" posee el valor **app-root**, gracias al cual el componente puede incluirse en nuestras páginas **HTML** (p. ej., en **index. html**) mediante el tag **<app-root></app-root>**

Los componentes se componen de tres partes:

1. Anotaciones.
2. Vista.
3. Controlador.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = '007 Componentes';
}
```

Las anotaciones son metadatos que agregamos a nuestro código y que permiten ciertas propiedades como son:

- ***Selector***.- Nombre a utilizar como tag HTML para indicar a Angular que cree una instancia del componente cuanto se encuentre con dicho tag.
- ***Template***.- Plantilla a utilizar para el componente y que se describe en el propio fichero ***ts*** del mismo.
- ***TemplanteUrl***.- Indica la URL en el que se halla la plantilla en caso de definirse externamente.
- ***Style***.- definición de estilos para la plantilla.
- ***StyleUrl***.- array que contiene los diferentes archivos de estilos que pueden utilizarse en el componente en curso.
- ***Host***.- permite encapsular ciertas partes del componente.
- ***Directive***.- array con las directivas que pueden utilizarse en el componente.
- ***Input***.- define variables que recogen información del padre.
- **Output**.- define variables que pasan información al padre.

```
1   import { Component, OnInit } from '@angular/core';
2
    You, a few seconds ago | 2 authors (You and others)
3   @Component({
4   selector: 'app-hello-world',
5   templateUrl: './hello-world.component.html',
6   styleUrls: ['./hello-world.component.css']
7   })
8
9   export class HelloWorldComponent implements OnInit {
10
11  constructor() { }
12
13  ngOnInit() {
14  }
15
16  }       You, a few seconds ago • Uncommitted changes
```

Una ***Vista*** contiene el template que usaremos para elaborar la pantalla asociada al componente. Esta pantalla podemos definirla en un archivo auxiliar o bien dentro del propio archivo donde definimos la clase controlador encerrando el código HTML entre apóstrofos (**‘<h1>{{title}}</h1>’**).

Un ***Controlador*** es la clase que contendrá la lógica del componente (definiciones, funciones, etc.)

Generalmente, un componente suele llevar asociados diversos archivos con el siguiente cometido:

- **app.component.html:** contiene el código HTML mediante el que fabricamos la pantalla (o **Vista**).
- **app.component.css:** define los estilos que usaremos en la vista del componente.
- **app.component.ts:** archivo que contiene la lógica del componente (**Controlador**) y que se escribe en **TypeScript** para ser traducido a **JavaScript** posteriormente.
- **app.component.spec.ts:** archivo usado para la realización de test unitarios.

Cuando se analice el ciclo de vida de un componente, veremos que es posible realizar ciertas inicializaciones al crear el componente en el método **ngOninit**.

Un componente puede tener atributos de entrada y/o de salida para intercambiar información con otros componentes padres e hijos. Dichos atributos requerirán del uso de las directivas **@Input()** y **@Output().**

Metadatos

Permiten decorar una clase y configurar así su comportamiento. Extiende una función mediante otra sin tocar la original. Los metadatos permiten configurar el comportamiento de una clase a la que se le ha asociado un decorador.

Por ejemplo, una clase no se trata como un componente hasta que no se le asocia el decorador **@Component** con su metadata necesaria. Los decoradores más comunes son **@Component** y **@Injectable**

Decorador	Propiedades metadata	Módulo a importar
@Input	bindingPropertyName: string	Input
@Output	bindingPropertyName: string	Output
@Attribute	atributeName: string	Attribute
@Hostlistener	args: string[] eventName: string	HostListener
@Host		Host
@Inject	token: any	Inject
@Injectable		Injectable

Decorador	Propiedades metadata	Módulo a importar
@Component	animation: any[] changeDetection ChangeDetectionStrategy encapsulation : ViewEncapsulation entryComponent: Array<type<any>|any[]> interpolation: [string, string] moduleId: string styles: string[] styleUrls: string[] template: string templateUrl : string viewProviders : Provider[]	Component

Decorador	Propiedades metadata	Módulo a importar
@Directive	exportAs : string host : {[key : string]: string} inputs : string[] outputs : string[] providers : Provider[] queries : {[key: string]: any} selector: string	Directive
@NgModule	bootstrap: Array<Type<any>|any[]> declarations: Array<Type<any>|any[]> entryComponents: Array<Type<any>|any[]> exports: Array<Type<any>|any[]> id: string imports: Array<Type<any> |ModuleWithProviders|any[]> providers: Provider[] schemas: Array<SchemaMetadata|any[]>	NgModule

Decorador	Propiedades metadata	Módulo a importar
@Optional		Optional
@Pipe	name: string pure: boolean	Pipe
@Self		Self
@SkipSelf		SkipSelf

- **Animations:** lista de animaciones del componente.
- **Bootstrap:** define los componentes que deberían tenerse en cuenta en el arranque cuando este módulo es Bootstrap.
- **ChangeDetection:** estrategia de detección de cambios utilizada por el componente.
- **Declarations:** lista de directivas/pipes pertenecientes al módulo.
- **Encapsulation**: estrategia de encapsulación de estilo usada por el componente.
- **EntryComponents:** lista de componentes que deben compilarse cuando se define este módulo.
- **ExportAs:** nombre usado para exportar la instancia del componente en una plantilla.
- **Exports:** lista de directivas, pipes y módulos que pueden usarse en otros componentes que tengan importado el módulo en curso.
- **Host:** mapa de propiedad de clase para enlaces de elementos de host para eventos, propiedades y atributos.
- **Id:** usado para identificar módulos en getModuleFactory.
- **Imports:** lista de módulos cuyas directivas/pipes exportadas pueden utilizarse en el módulo.
- **Inputs:** lista de nombres de propiedades de clase para enlazar datos como entradas de componentes.
- **Interpolation:** marcadores de interpolación personalizados utilizados en la plantilla del componente.
- **ModuleID:** ID del módulo ES/CommonJS del archivo donde se define el componente.
- **Outputs:** lista de nombres de propiedades de clase que exponen eventos de salida a los que otros pueden suscribirse.
- **PreserveWhitespaces:** permite mantener o eliminar espacios en blanco.
- **Providers:** define el conjunto de objetos

- **Queries:** consultas que pueden inyectarse en el componente.
- **Schemas:** usado para declarar elementos y propiedades que no son componentes ni directivas Angular.
- **Selector:** tag con el que se identifica el componente en una plantilla.
- **Styles:** estilos “en línea” que hay que aplicar a la vista del componente.
- **StyleUrls:** lista de URL con hojas de estilo que pueden aplicarse a la vista del componente.
- **Template:** plantilla “en línea” aplicada a la vista del componente.
- **TemplateUrl:** URL que apunta al archivo que contiene la vista del componente.
- **ViewProviders:** lista de proveedores disponibles para el componente e hijos.

data binding

Permite intercambiar datos entre el template y la clase del componente que soporta la lógica del mismo. Existen 4 tipos de **Data binding** (que analizaremos en profundidad más adelante) y que son:

- **Interpolación:** muestra el valor de una propiedad en el lugar donde se incluya **{{ propiedad }}**.
- **Property binding:** traspaso del objeto del componente padre al componente hijo**[objHijo]="objPadre"**).
- **Event binding:** permite invocar un método del componente pasándole un argumento al producirse un evento (p. ej., **(click)="hazAlgo(obj)"**).
- **Two-way binding:** binding bidireccional que permite combinar **event** y **Property binding** (p. ej., **input [(ngModel)]= "obj"**).

Directivas

Permite añadir comportamiento dinámico a HTML mediante una etiqueta o selector. Existen directivas estructurales (***ngFor** o ***ngIf**) o de atributos que modifican el aspecto o comportamiento de un elemento **DOM** (**NgSwitch**, **NgStyle** o **NgClass**).

Servicios

Son clases que permiten realizar acciones concretas que se ponen a disposición de cualquier componente. Por ejemplo, permiten obtener datos, resolver lógicas especiales o realizar peticiones a un servidor.

Dependency injection

Permite suministrar funcionalidades, servicios, etc., a un componente sin que sea el propio componente el encargado de crearlos. Utiliza el decorador **@Injectable()**. Generalmente proporciona servicios y facilita los test y la modularidad del código.

Angular CLI

Angular CLI es una herramienta que permite crear aplicaciones Angular desde el terminal apoyándose en plantillas de código

Entre las funcionalidades que obtendremos al usar Angular CLI destacan las siguientes:

- Herramientas para **testing**.
- Herramientas para **despliegue**.
- **Servidor HTTP** para lanzar el proyecto.
- Sistema **live-reload** que permite replicar de forma inmediata en el navegador cualquier cambio detectado en el proyecto.

Comandos

Comando	Descripción
help	Invoca a la ayuda.
new	Permite crear una nueva aplicación.
serve	Construye la aplicación y arranca el servidor web.
generate	Permite generar los siguientes elementos: class, component, directive, enum, guard, interface, module, pipe, service, etc.
lint	Permite usar tslint (herramienta que permite revisar el código durante su edición).
Test	Permite generar un directorio de salida para pruebas.
e2e	Sirve la aplicación y ejecuta pruebas.
build	Compila la aplicación en un directorio de salida.
get/set	Obtiene/establece un valor de la con?guración.
doc	Abre la documentación oficial Angular para buscar información sobre la palabra indicada.
eject	Permite extraer el archivo de configuración de webpack.
xi18n	Extrae los mensajes i18n de las plantillas

Actividad

1. Abriremos una ventana de **CMD** y nos situaremos en el directorio **C:\Ej100_angular** para teclear el siguiente comando: **`ng new HMAC`**

Would you like to add Angular routing? (y/N): esto es para agregar el módulo de rutas dentro de nuestro aplicativo, para agregarlo hay que escribir y + Enter. En caso de no agregarlo, siempre se puede agregar manualmente a la aplicación.

Which stylesheet format would you like to use?: esta opción es para definir el motor de CSS que vamos a utilizar, puede ser CSS, SASS, LESS o Stylus.

- Observaremos que, al final de la creación, aparece un mensaje que indica que el proyecto se ha creado satisfactoriamente.
- Seguidamente, abriremos el proyecto con nuestro editor
- Una vez abierto el proyecto, modificaremos el archivo **app.component.ts** ubicado en **HMAC/src/app** y a la variable **title** le asignaremos el valor '**Hola Mundo Angular CLI**'.
- Podemos modificar también el fichero **html** que se crea por defecto ubicado en **HMAC/src/app/app.component.html** para dejarlo simplemente con el siguiente contenido:

```
<div style="text-align:center">
  <h1>
    {{title}}!
  </h1>
</div>
```

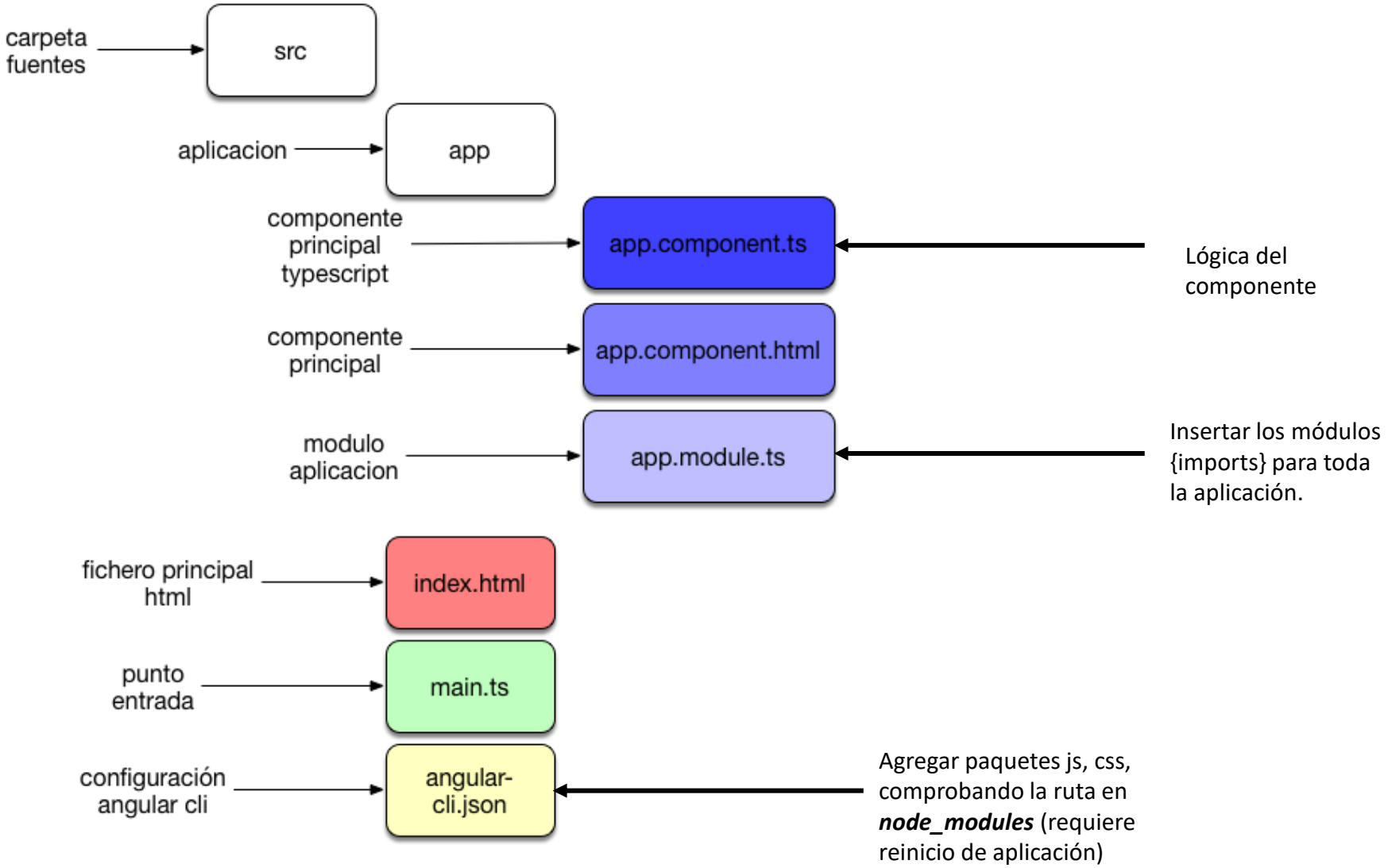
2. Una vez salvado el archivo, pasaremos a ejecutar la aplicación (utilizando el comando **serve**) accediendo al directorio del proyecto y tecleando lo siguiente: **`ng serve`** (dentro de la carpeta recién creada HMAC)

3. Observaremos que, tras una serie de mensajes, la aplicación arranca y recibimos un mensaje indicando que la compilación se ha realizado satisfactoriamente.

4. Por último, abriremos nuestro navegador y comprobaremos el resultado tras acceder al siguiente URL:

http://localhost:4200/

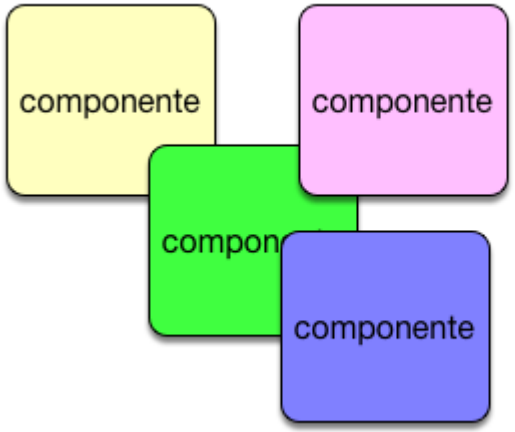
5. De esta aplicación podemos señalar los ficheros fundamentales



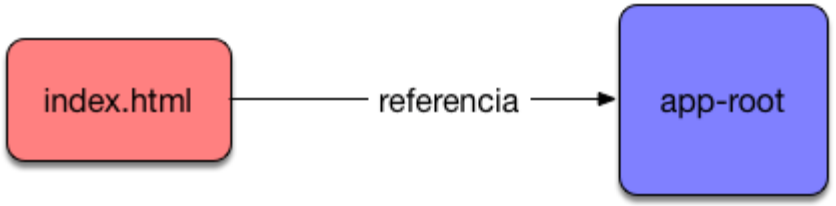
El primer fichero que destaca es el de index.html. Este es el fichero de entrada a la aplicación y contiene muy poco código . El código más importante que contiene este ubicado en el body.

<body>
<app-root></app-root>
</body>

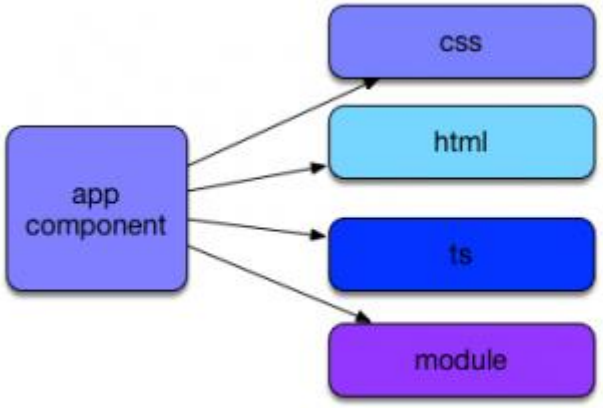
Como podemos ver el body dispone de una etiqueta especial ***<app-root></app-root>*** esta etiqueta es la encarga de definir el componente principal de Angular. Angular es un framework que tiene un fuerte enfoque de construcción de componentes visuales con TypeScript.



Así pues, ya tenemos la primera parte, el fichero de Index que referencia al componente raíz.



¿Donde tenemos el componente Raíz? .El componente raíz se encuentra ubicado en la carpeta app y se denomina a nivel de fichero app.component.



Angular divide el componente en varias partes

- ***app.component.css***: Define el estilo del componente
- ***app.component.html***: Define la plantilla del componente
- ***app.component.ts***: Define la funcionalidad de TypeScript del componente.
- ***app.module.ts***: Define el módulo al que pertenece el componente.

Es en el fichero app.component.ts en donde Angular enlaza la etiqueta <app-root> con el nombre del componente:

```
import { Component } from '@angular/core';

@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'app';
}
```

El fichero *main.ts*

Ya tenemos bastante claro como se relacionan los componentes y la página index.
Ahora bien ¿para que sirve el fichero ***main.ts***?.

Este fichero es el encargado de definir que módulo es el de arranque.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
.catch(err =&amp;amp;amp;gt; console.log(err));
```

Podemos ver como solicita que se cargue ***AppModule***

El fichero *main.ts*

Ya tenemos bastante claro como se relacionan los componentes y la página index.
Ahora bien ¿para que sirve el fichero ***main.ts***?.

Este fichero es el encargado de definir que módulo es el de arranque.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
enableProdMode();
}

platformBrowserDynamic() bootstrapModule(AppModule)
.catch(err =&amp;amp;amp;gt; console.log(err));
```

Podemos ver como solicita que se cargue ***AppModule***

El fichero de ***app.module.ts*** se encuentra en la carpeta src. Vamos a mostrar su contenido:

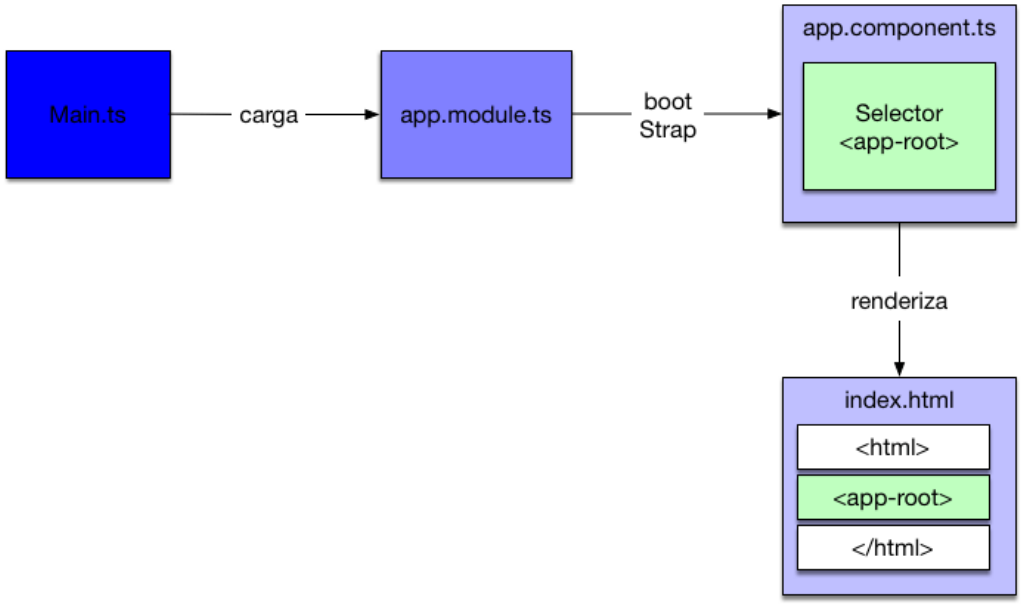
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
declarations: [
AppComponent
],
imports: [
BrowserModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Como podemos ver es este fichero el que a través de la propiedad de ***bootstrap*** nos indica qué componente es el de arranque, en este caso ***AppComponent***.

Así lo que sucede en una aplicación Angular es, que, en un primer momento, se carga el fichero ***main.ts*** que define cual es el modulo de arranque.

Angular accede al módulo de Arranque y busca cual es el componente inicial que arrancar y lo arranca.

Este componente dispone de un decorador ***"selector":app-root*** que indica al framework que etiqueta le identifica. Esta es la etiqueta que se encuentra en el fichero index.html.



AJAX

La Base de la Web Actual

Primero será bueno repasar las tecnologías estándar que de las que disponemos en todos los navegadores.

El lenguaje HTML (HyperText Markup Language)

HTML es el lenguaje básico con el que podemos crear páginas web, con el paso del tiempo se han ido añadiendo etiquetas a las que ya tenía además de dar soporte a otros lenguajes como CSS (Cascade Style Sheets).

El lenguaje XHTML (eXtensible HTML)

Este lenguaje es el que ha supuesto el mayor cambio desde 1997 (HTML 3.2), ya que busca proporcionar páginas web ricas en contenido a un amplio abanico de dispositivos PC, Móviles y dispositivos con conexión inalámbrica.

CSS (Cascade Style Sheets)

En los primeros tiempos de las páginas web, se tenía en un mismo documento ".html" tanto los párrafos de texto e imágenes como su estilo, e indicábamos el tipo de atributos del texto dentro de las etiquetas HTML.

XML (eXtensible Markup Language)

XML es un metalenguaje que fue ideado para "describir" un conjunto de datos como pudiera ser los campos de una tabla para intercambiar información de forma rápida y fácil. También permite especificar cómo tienen que ser los datos, por lo que se decidió especificar el lenguaje HTML con XML y nació XHTML.

JSON (JavaScript Object Notation)

Es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos.

DOM (Document Object Model)

Cuando se carga una página web en el navegador, éste último crea asociado a la página una serie de objetos, de manera que mediante un lenguaje utilizable en la parte del navegador(el cliente), como JavaScript, pueden modificarse las características de esos objetos y con ello la página en sí.

Lenguajes de Cliente, (JavaScript)

Cuando el usuario ve una página web en su navegador, ésta puede tener, aparte del código HTML, código escrito en un lenguaje de script que se utiliza normalmente para dotar de dinamismo a las páginas, obteniendo DHTML.

Lenguajes de Servidor (JSP, PHP, Ruby, C#, VB, etc.)

A veces necesitamos enviar información al servidor y que éste la procese y nos responda (por ejemplo, al conectarse a nuestra cuenta de correo), o que guarde información (por ejemplo, en un foro). Para este procesamiento se utilizan los lenguajes de servidor PHP, JSP (el que utiliza este texto), etc. Puedes elegir el que más te guste con sus pros y sus contras,

Visión en Conjunto

Tendremos un usuario que carga en su navegador una página compuesta por XHTML y JavaScript cuyo estilo está en un archivo CSS si lo separamos; el navegador crea el DOM asociado a la página y el JavaScript lo modifica para conseguir dinamismo.

Tenemos en el servidor páginas hechas con JSP, cuando el usuario pide una de estas páginas, el servidor la ejecuta y devuelve el resultado al usuario ya sea una página XHTML u otro tipo de información.

Las RIA (Rich Internet Application Technologies)

Para que entendamos la necesidad de uso de AJAX, vamos a ver una serie de términos, problemas y posibles soluciones y ver cuál es el papel de AJAX dentro de todo esto.

Características de una aplicación de escritorio

Si le echamos un vistazo a una aplicación típica de escritorio vemos que tiene las siguientes cualidades:

- Responde de forma intuitiva y rápida.
- Da respuesta inmediata a los actos del usuario.

Características de una aplicación Web Convencional:

- Cada vez que pulsamos sobre un link, esperamos y la página se refresca.
- La página refresca todos los eventos, envíos y datos de la navegación.
- El usuario debe esperar la respuesta.
- Modelo de petición/respuesta de comunicaciones síncrono.
- El estado del trabajo que estamos desarrollando se basa en qué página estamos.

Problemas de una aplicación web convencional:

- Respuesta lenta.
- Pérdida del contexto durante el refresco.
- Perdemos información en la pantalla que habíamos rellenado.
- Perdemos la posición del scroll de la pantalla.
- No tenemos respuesta inmediata a nuestros actos.
- Tenemos que esperar que llegue la siguiente página.

DHTML = HTML + Javascript + DOM + CSS

POSITIVO

- o Se utiliza para crear aplicaciones interactivas y más rápidas.

NEGATIVO

- o La comunicación es ***síncrona***.
- o Requiere el refresco completo de la página, perdiendo parte del contexto.

AJAX = DHTML + XMLHttpRequest

Es un refinamiento del DHTML, utiliza todas sus herramientas, sumándole el objeto ***XMLHttpRequest*** para obtener información de manera ***asíncrona*** y refrescar solo la parte necesaria de la página sin perder nada del contexto, se terminaron los problemas del DHTML.

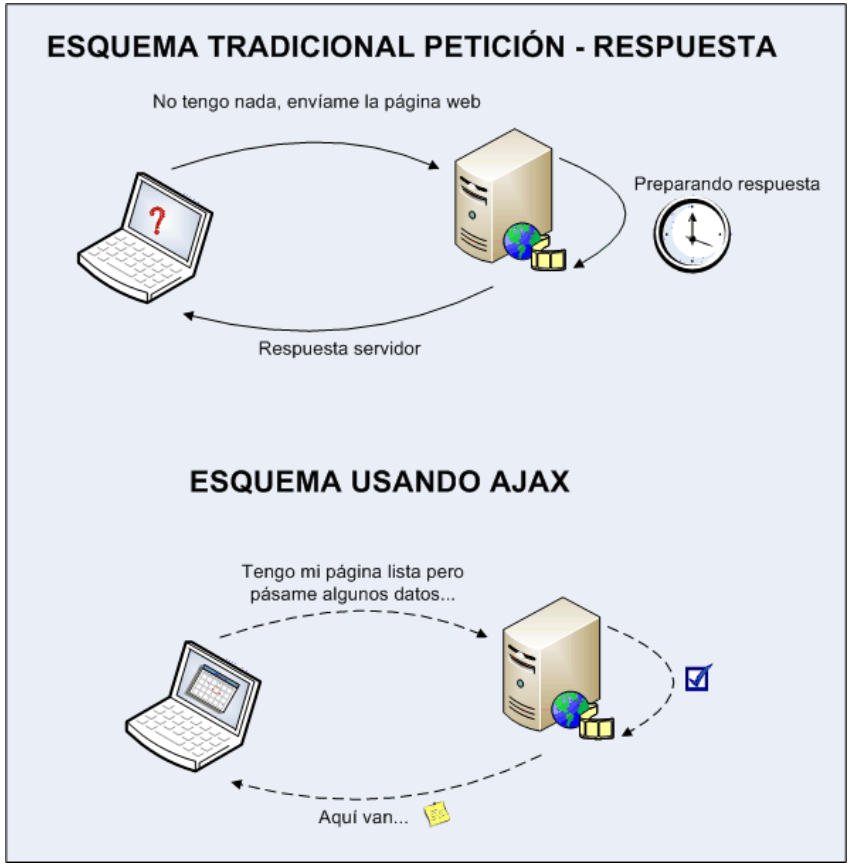
Algunos ejemplos reales de AJAX

Lo mejor es ver algunas posibilidades del uso de AJAX, se va a hacer mención primero a la primera aplicación AJAX conocida, Microsoft inventó el objeto que más tarde se convertiría en el XMLHttpRequest y lo usó en su versión web del ***Outlook,*** Otro gran ejemplo del uso de AJAX es ***Google maps***

Fundamentos de AJAX

(Asynchronous JavaScript And Xml)

AJAX es un estilo de programación que permite al desarrollador hacer que una página web interactúe con el servidor web que lo aloja. Si utilizamos AJAX, una vez que la página se ha cargado puede ejecutar comando en el servidor, recibir cualquier información resultante de dichos comandos y mostrarlos al usuario sin necesidad de recargarse.



JavaScript

```html
<html>
<head>
<tilte></title>
<script>
function mensaje(){
    alert('Hola que tal');

}
</script>
</head>
<body>

    <a href="#" onClick = "mensaje();">Haz click aqui</a>

</body>
</html>

```

JavaScript es un lenguaje de programación que se ejecuta del lado del cliente y que permite escribir funciones como parte de un archivo HTML, con JavaScript se pueden realizar cálculos complejos y mostrar el resultado en la página web.

Con AJAX, JavaScript puede comunicarse con el servidor web y mostrar la información resultante.

HMLHttpRequest

Es un objeto JavaScript, diseñado por Microsoft y adoptado por Mozilla, Apple y Google. Proporciona una forma fácil de obtener información de una URL sin tener que recargar la página completa

Métodos	Descripción
open(método, URL, banderaAsync, nombreuser, password)	Según el método (GET o POST) y la URL se prepara la petición. Si la banderaAsync=true Petición asíncrona, si es fase la petición es síncrona. nombreuser y password solo se usan para acceder a recursos protegidos.
send(contenido)	Ejecuta la petición, donde la variable contenido son datos que se envían al servidor.
abort()	Para la petición que está procesando.
getAllResponseHeaders()	Devuelve todas las cabeceras de la llamada HTTP como un string.
getResponseHeader(cabecera)	Devuelve la cabecera identificada por la etiqueta.
setRequestHeader(etiqueta, valor)	Establece el valor de una etiqueta de las cabeceras de petición antes de que se haga la petición.

Propiedades	Descripción
status	Código devuelto por el servidor
statusText	Texto que acompaña al código
responseText	Datos devueltos formato string
responseXML	Datos devueltos formato Objeto XML
readyState	Estado actual de la petición. 0: Sin iniciar 1: Cargando 2: Cargado 3: Interactivo (algunos datos devueltos) 4: Completado
onreadystatechange	Puntero a la función del manejador que se llama cuando cambia readyState.

open(method, URL, async)

Este método prepara el objeto para la transmisión de información al servidor. Se puede definir con una serie de tipos, como **POST**, **GET** y **PUT**

La ***URL*** contiene la aplicación a donde se enviara (o recibirá) la información y la propiedad ***async***, puede ser **true** o **false**, si es false, indica que el objeto debe esperar a recibir una respuesta del servidor antes de continuar procesando el código JavaScript, si es true se puede procesar el código mientras se espera la respuesta.

setRequestHeder(label, value)

Este método permite indicar al objeto qué tipo de texto le enviaremos, especificándolo en el encabezado HTTP, **label** sele ser casi siempre “Content-type” y se pueden elegir una serie de tipos, incluidos los formatos XML y HTML o un conjunto de resultados de un formulario

send(content)

Este método trasmite información textual al servidor.

abort()

Detiene la petición que se está procesando

readyState

El valor de esta propiedad se modifica en función del estado del objeto XHR:

Será 0 si no está inicializada

Será 1 si está abierta

Será 2 si está enviada

Será 3 si se recibe

Será 4 si está cargada

onreadyStateChange

Esta propiedad provoca una acción cada vez que el readyState del objeto cambia.

responseText

Esta propiedad toma el valor de la información textual que devuelve el servidor.

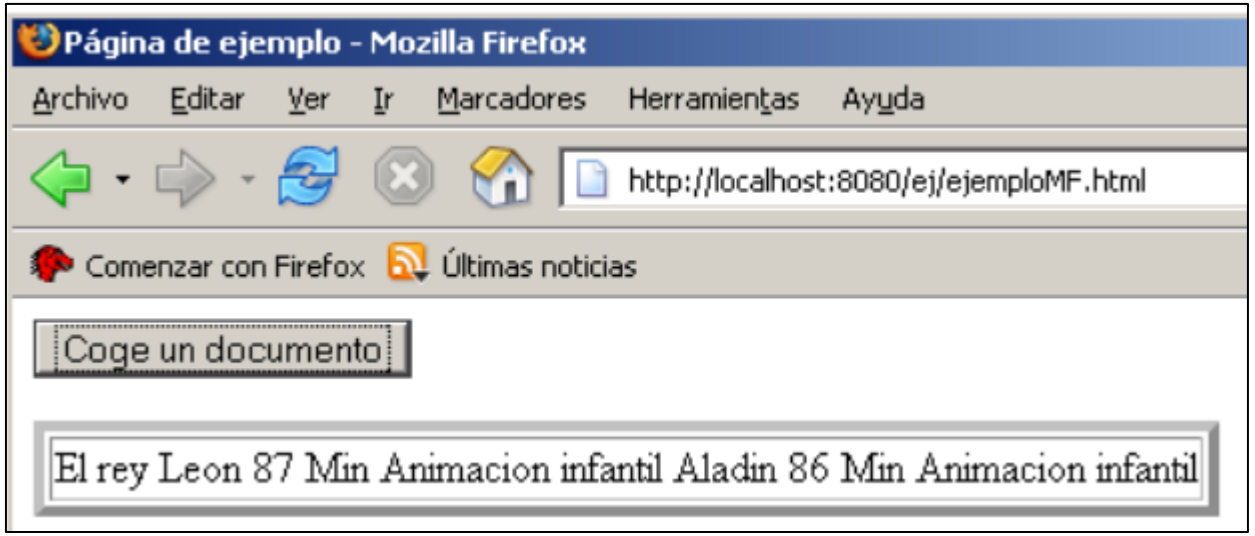
Por medio de JavaScript, preparamos el objeto XHR con toda la información necesaria, lo enviamos y recibimos los resultados en la propiedad ***resposeText***. A continuación, utilizamos el método ***getElementById*** de JavaScript para insertar el resultado en alguna parte del documento.

Archivo “datos/videoclub.xml”

```xml
<?xml version="1.0" encoding="UTF-8"?>
  <VideoClub>
            <Pelicula>
              <Titulo>El rey Leon</Titulo>
              <Duracion>87 Min</Duracion>
              <Genero>Animacion infantil</Genero>
            </Pelicula>
            <Pelicula>
              <Titulo>Aladin</Titulo>
              <Duracion>86 Min</Duracion>
              <Genero>Animacion infantil</Genero>
            </Pelicula>
  </VideoClub>
```

Archivo “ejemplo.html”

```
<html>
<head>
  <title>Ejemplo Ajax</title>
   <script language="JavaScript" type="text/javascript">
                var peticion01 = null; //Creamos la variable
                peticion01 = new XMLHttpRequest();

                function Datos(url) {
                                if(peticion01) {
                                                peticion01.open('GET', url, false);
                                                peticion01.send(null);
                                                document.getElementById('resultado').innerHTML = peticion01.responseText;
                                                    }
                                            }
    </script>
</head>
<body>
                <button onclick=“Datos('datos/videoclub.xml')">Coge un documento</button>
                <table border="4">
                 <tr><td><span id="resultado">Sin resultado</span></td></tr>
                </table>
</body>
</html>
```


Página de ejemplo - Mozilla Firefox
Archivo Editar Ver Ir Marcadores Herramientas Ayuda
http://localhost:8080/ej/ejemploMF.html
Comenzar con Firefox
Últimas noticias
Coge un documento
El rey Leon 87 Min Animacion infantil Aladin 86 Min Animacion infantil

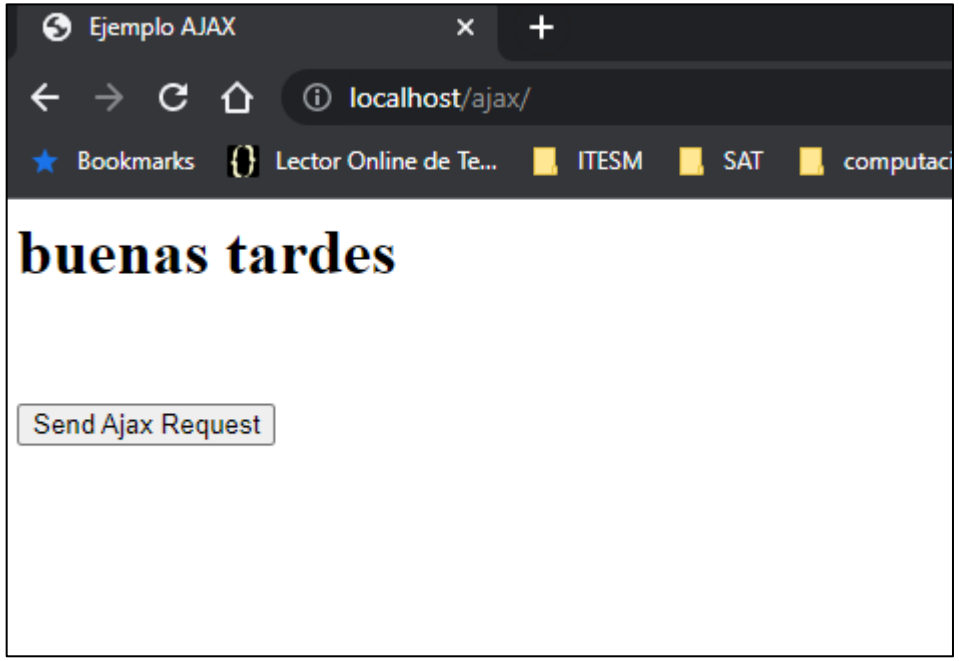
EJEMPLO. TECLEAR EL ARCHIVO INDEX.HTML

```html
<html>
<head>
<tilte>Ejemplo AJAX</title>
<script>
function sendRequest(){

        var Objeto = new XMLHttpRequest();
        Objeto.open('GET', 'backend.php', true);
        Objeto.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
        Objeto.onreadystatechange = function() {
            document.getElementById('title').innerHTML = Objeto.responseText;
        }
        Objeto.send();

}
</script>
</head>
<body>

    <h1 id='title'> </h1>   <br><br>

    <button onClick="sendRequest();" >Send Ajax Request</button>

</body>
</html>
```

hace consulta, guarda info

EJEMPLO. TECLEAR EL ARCHIVO BACKEND.PHP

```php
<?php

date_default_timezone_set("America/Monterrey");

$hora = getdate();

if($hora['hours'] < 12)
    echo "buenos días";
else{
if($hora['hours'] < 20)
    echo "buenas tardes";
else
    echo "buenas noches";

}
?>
```


Ejemplo AJAX
localhost/ajax/
Bookmarks
Lector Online de Te...
ITESM
SAT
computaci
buenas tardes
Send Ajax Request

sistema de routing

Crear nuestra primera aplicación con rutas

1.- Importar el código del sistema de routing

En nuestro módulo principal tenemos que hacer los correspondientes ***imports***.

Deberías incluir a RouterModule y Routes, que obtenemos de la librería "***@angular/router***". El código será como este:

```
import { Routes, RouterModule } from '@angular/router';
```

RouterModule es el módulo donde está el sistema de rutas, por tanto, contiene el código del propio sistema de rutas.

Routes es una declaración de un de tipo, que corresponde con un array de objetos ***Route***. Los objetos ***Route*** están declarados por medio de una interfaz en el sistema de routing. Esta interfaz sirve para que en la declaración de tus rutas coloques solamente aquellos valores que realmente son posibles de colocar. Si no lo haces así, el compilador de TypeScript te ayudará mostrando los correspondientes errores en tiempo de desarrollo.

2.- Crear la lista de las rutas disponibles en la aplicación

En este paso tenemos que crear un array de las rutas que queremos generar en nuestra aplicación. Utilizaremos la declaración del tipo ***Routes***, que hemos dicho es un array de objetos Route.

Cada objeto Route del array tiene un conjunto de campos para definir la ruta, definido por la interfaz Route, siendo que lo general es que tenga al menos un camino "path" y un componente para representar en esa ruta.
El código que producirás será más o menos así.

```
const rutas: Routes = [
  { path: '', component: HomeComponent },
  { path: 'contacto', component: ContactoComponent }
];
```

3.- Declarar el sistema de routing en el "imports" del decorador *@NgModule*

Ahora hay que editar el decorador del módulo principal ***@NgModule***, donde tendrás que colocar en el array "imports" la declaración de que vas a usar el sistema de routing de Angular.

Tenemos que indicar a RouterModule, pero en ese imports necesita la configuración de rutas, creada en el array Routes[] del paso anterior.

De momento, esa configuración la cargamos mediante un método forRoot() en el que pasamos por parámetro el array de rutas.

Se ve mejor con el código. Este es el decorador completo, pero te tienes que fijar especialmente en el array "imports".

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ContactoComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(rutas)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

4.- Definir el template del componente principal

Comenzamos el trabajo en el componente principal. En este componente tenemos que meter un poco de código en la vista, archivo "app.component.html".

En el template se crea el navegador de las rutas, indicando en los enlaces la directiva "routerLink", con el valor de cada ruta.

```
<nav>
  <a routerLink="/">Home</a> |
  <a routerLink="/contacto">Contacto</a> |
</nav>
```

No puede haber ningún enlace generado, que no tenga su correspondencia en el sistema de rutas. Fíjate que los enlaces comienzan por una "/" para hacerlos absolutos a la raíz del dominio.

Por último, debes colocar un contenedor llamado "router-outlet" para definir el espacio donde se van a inyectar los componentes. Es tan sencillo como colocar esta etiqueta definida por el módulo del sistema de routing.

```
<router-outlet></router-outlet>
```

Problemas comunes con el sistema del routing

Lo que hemos hecho hasta ahora es poco, por lo que no debería haber muchos problemas con nuestro ejemplo, pero vamos a ver un par de posibles errores que nos podemos encontrar.

Colocar un enlace a una ruta inexistente
Si nos equivocamos al escribir en un enlace la ruta, obtendremos un mensaje de error al hacer clic sobre ese enlace.
Por ejemplo, escribimos "contato" en vez de "contacto" en la ruta.

```
<a routerLink="/contato">Contacto</a>
```

Este es el error que vamos a encontrar: *Error: Cannot match any routes. URL Segment: 'contato'*

Colocar un componente no existente o no accesible en el módulo

Hasta ahora las rutas estaban asociadas a un componente. Puede ocurrir que en la declaración del array de Routes indiques un componente que no existe, ya sea porque has escrito mal el nombre de la clase del componente, o porque estás intentando usar un componente que no está en este módulo o que no está importado por este módulo.

Por ejemplo, podemos colocar el nombre de un componente que no existe como ruta:

```
{ path: '', component: HComponent },
```

En este caso el editor probablemente te alertará del error, señalando con una línea en rojo, pero si no lo ves, el compilador de TypeScript será el que te avise.

El mensaje de error que recibiremos será algo como: Cannot find name 'HComponent'.

Eso era porque HComponent no existía, pero podría ocurrir que el componente sí que exista, lo hayas importado, pero no esté disponibe en el módulo principal, entonces el error sería algo como *"CompNoImportadotComponent is not part of any NgModule or the module has not been imported into your module"*.

Equivocarse en la ruta en la declaración de rutas

Otra cosa que te puede ocurrir es que coloques una "/" al pricipio de una ruta. Fíjate que en la configuración de las rutas la ruta raíz es "" (cadena vacía) y no "/".

Esto sería incorrecto:

```
const rutas: Routes = [
  { path: '/', component: HomeComponent },
  { path: '/contacto', component: ContactoComponent }
];
```

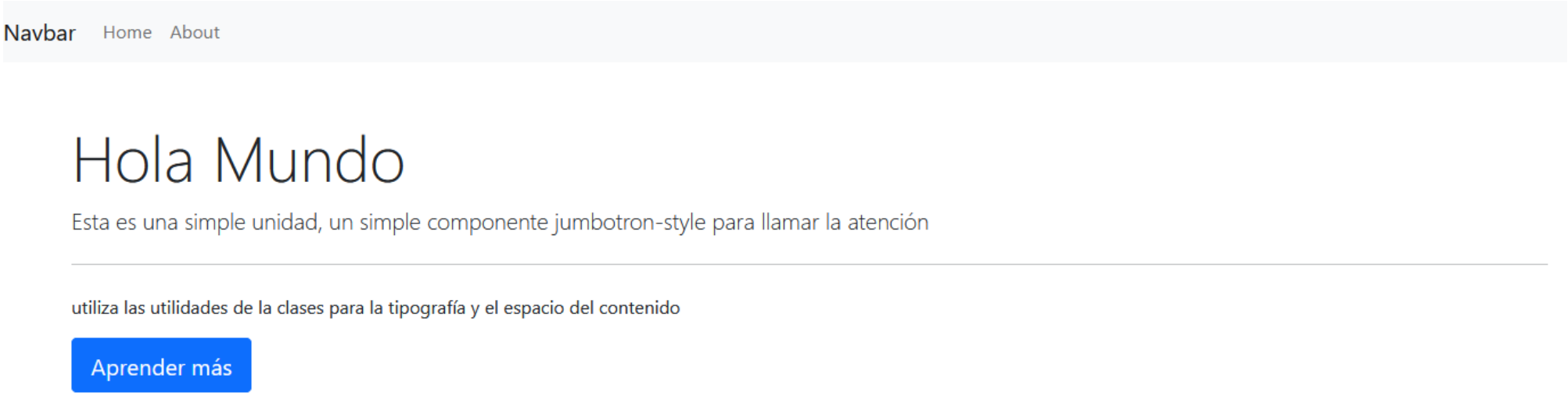
Para advertirte que las rutas no comienzan por "/", Angular te mostrará el siguiente error en la consola del navegador: *"Invalid configuration of route '/': path cannot start with a slash"*.

Actividad (Caso de estudio)

- Crear un proyecto sencillo en Angular
- Entender el uso de rutas en Angular
- Usar el componente **Router** para la carga dinámica de páginas
- Revisar el correcto funcionamiento del proyecto sencillo
- Si hay un problema, determinar su corrección
- Revisar nuevamente el correcto funcionamiento del proyecto sencillo

1. Creación de los elementos básicos del proyecto

El proyecto que vamos a crear está formado inicialmente por dos páginas (home y about) e incluirá una barra de navegación en la parte superior. Usaremos también Bootstrap, lo que le dará un aspecto similar al de la figura:



Desde una terminal comenzamos creando el proyecto con el CLI de Angular

ng new miapp
cd miapp

1.1. Instalación de Bootstrap

La opción que elegiremos para incluir *Bootstrap* en nuestro proyecto será usarlo como un paquete más del proyecto siguiendo estos pasos. En primer lugar, instalamos los paquetes necesarios (*bootstrap, jquery* y *@popperjs/core*) y usaremos la opción *--save* para guardar sus dependencias en el archivo *package.json*. Esto hará que estos paquetes se instalen cuando otros utilicen el proyecto e instalen las dependencias con *npm install*.

npm install bootstrap jquery @popperjs/core --save

Incluir en la sección architect.build de angular.json las referencias al CSS y los JS. Esto es equivalente a incluir en el index.html el CSS y los JS de Boostrap.

```
"styles": [ "src/styles.css",
"node_modules/bootstrap/dist/css/bootstrap.min.css" ],
"scripts": [
"node_modules/jquery/dist/jquery.min.js",
"node_modules/@popperjs/core/dist/umd/popper.min.js",
"node_modules/bootstrap/dist/js/bootstrap.min.js" ]
```

viernes 2 sep 22
1.27.03

1.2. Creación de los componentes

Organizaremos los componentes y los servicios de la aplicación en carpetas:

- Una carpeta components para los componentes.
- Una carpeta componentes/shared para componentes compartidos (p.e. la barra de navegación)
- Una carpeta services para servicios que veremos más adelante cuando creemos el primer servicio.

Creamos los componentes con el CLI de Angular

ruta / nombre

```
ng generate component components/home
ng generate component components/about
ng generate component components/shared/navbar
```

Los componentes generados son añadidos de forma automática al array declarations en app.module.ts

```
@NgModule({
declarations: [
AppComponent,
HomeComponent,
AboutComponent,
NavbarComponent ],
```

Archivos generados al crear un componente

De forma predeterminada, al crear un componente se generan 4 archivos:
- Un CSS para estilos específicos a aplicar únicamente en el componente (.css).
- Un HTML para la vista del componente (.html).
- Un archivo TypeScript para programar la lógica del componente (.ts).
- Un archivo para las pruebas (.spec.ts).

Las aplicaciones Angular se desarrollan como un conjunto de componentes que pueden anidarse. En el archivo TypeScript se especifica qué etiqueta se debe utilizar para usar el componente en HTML. Esta etiqueta es lo que se conoce como *selector*.
A continuación, se muestra el código TypeScript de la barra de navegación en el que se puede apreciar el selector generado por el CLI de Angular.

```
import { Component, OnInit } from '@angular/core';

@Component({
selector: 'app-navbar',
templateUrl: './navbar.component.html',
styleUrls: ['./navbar.component.css'] })

export class NavbarComponent implements OnInit { constructor() { }

ngOnInit():  void {
   }
}
```

1.3. Implementación de la página home

Sustituimos el contenido de ***components/home.component.html*** por un jumbotron de Bootstrap.

```
<div class="jumbotron">
    <h1 class="display-4">Hola Mundo</h1>
    <p class="lead">
        Esta es una simple unidad, un simple componente jumbotron-style para llamar la
atención
        </p>
    <hr class="my-4" />
    <p>
        utiliza las utilidades de la clase para la tipografía y el espacio del
contenido
    </p>
    <a class="btn btn-primary btn-lg" href="#" role="button">Aprender más</a>
 </div>
```

1.4. Implementación de la barra de navegación

Sustituimos el contenido de ***components/shared/navbar.html*** por un ***navbar*** sin cuadro de búsqueda de Bootstrap. En las opciones de menú dejamos sólo dos elementos (uno para ***home*** y otro para ***about***)

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Navbar</a>
    <button
        class="navbar-toggler"
        type="button"
        data-toggle="collapse"
        data-target="#navbarNav"
        aria-controls="navbarNav"
        aria-expanded="false"
        aria-label="Toggle navigation"
    >
    <span class="navbar-toggle-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
            <li class="nav-item active">
                <a class="nav-link" href="#">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">About</a>
            </li>
        </ul>
    </div>
</nav>
```

1.5. Enrutado

Las rutas permitidas en nuestra aplicación estarán definidas en el archivo ***app-routing.module.ts*** (app.routes.ts) Inicialmente, definiremos una ruta para el ***home*** y otra para el ***about***. Más adelante, conforme vayamos incorporando funcionalidades a la aplicación añadiremos más rutas a este archivo.

```
import { HomeComponent } from './components/home/home.component’;
import { AboutComponent } from './components/about/about.component’;

export const ROUTES: Routes = [
{ path: 'home', component: HomeComponent },
{ path: 'about', component: AboutComponent },
{ path: '', pathMatch: 'full', redirectTo: 'home' },
{ path: '**', pathMatch: 'full', redirectTo: 'home' },
];
```

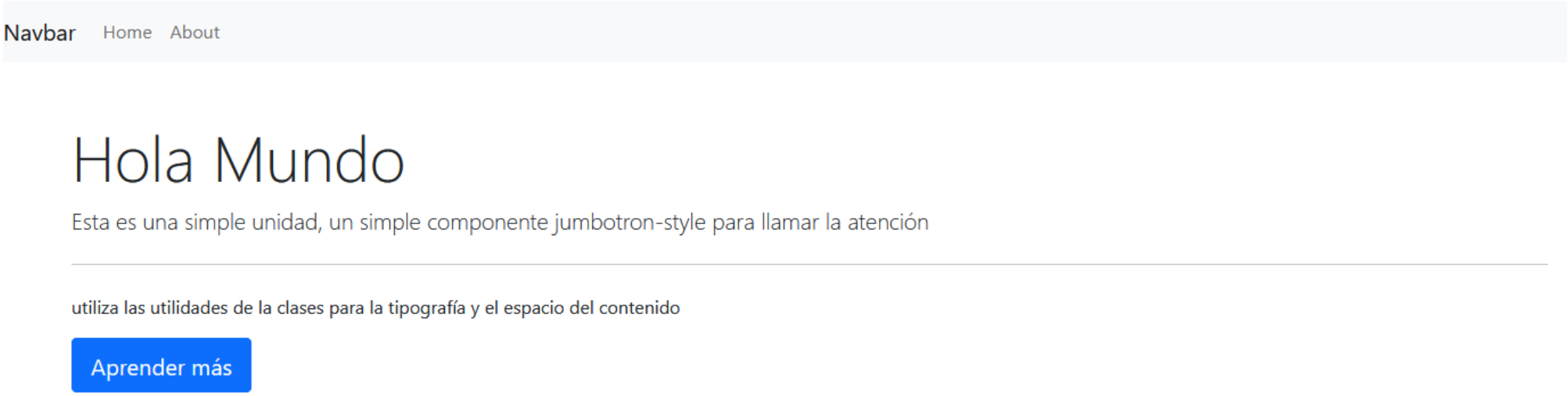
1.6. Modificar la página de la aplicación

La página de la aplicación se construye mediante la inclusión de componentes existentes. En el app.component.html dejamos sólo los selectores de los componentes que incluiremos en la página (la barra de navegación y el componente seleccionado desde la barra de navegación).

```
<app-navbar></app-navbar>
<div class="container m-5">
  <router-outlet></router-outlet>
</div>
```

1.8. Servir la aplicación

Desde el directorio de la aplicación con ***ng serve --open*** se ejecuta la aplicación y se abre en un navegador al guardar los cambios.



Introducción al sistema de Routing en Angular

Para facilitar la navegación por un sitio donde realmente sólo hay un index, existe lo que llamamos el sistema de routing, que tiene el objetivo de permitir que en el sitio web haya rutas internas, respondiendo a rutas "virtuales" como las que existen en los sitios tradicionales.

Llamamos "virtuales" a esas rutas, porque realmente sólo existe un "index.html", no habrá un archivo "contacto.html" o "productos.html" para cada ruta, sino que será realmente siempre el "index.html" el que se entregue al navegador.

El sistema de routing es el encargado de reconocer cuál es la ruta que el usuario quiere mostrar, presentando la pantalla correcta en cada momento. Esto es útil por varios motivos, entre ellos:

1. Permite que la aplicación responda a rutas internas. Es decir, no hace falta entrar siempre en la pantalla principal de la aplicación y navegar hasta la pantalla que queremos ver realmente.
2. Permite que el usuario pueda usar el historial de navegación, yendo hacia atrás y hacia adelante con el navegador para volver a una de las pantallas de aplicación que estaba viendo antes.

Angular, como un buen framework, dispone de un potente sistema de routing para facilitar toda la operativa de las single page applications.

Está compuesto por varios actores que tienen que trabajar juntos para conseguir los objetivos planteados.

Elementos básicos del sistema de routing que son necesarios para comenzar a trabajar.

•El módulo del sistema de rutas: llamado ***RouterModule***.

•Rutas de la aplicación: es un array con un listado de rutas que nuestra aplicación soportará.

•Enlaces de navegación: son enlaces HTML en los que incluiremos una directiva para indicar que deben funcionar usando el sistema de routing.

•Contenedor: donde colocar las pantallas de cada ruta. Cada pantalla será representada por un componente.

1.7. Incluir las rutas en la barra de navegación (Respuesta)

Para cada elemento de la barra de navegación incluiremos su ruta correspondiente definida previamente en ***app.routes.ts***

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">Navbar</a>
    <button
        class="navbar-toggler"
        type="button"
        data-toggle="collapse"
        data-target="#navbarNav"
        aria-controls="navbarNav"
        aria-expanded="false"
        aria-label="Toggle navigation"
    >
    <span class="navbar-toggle-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
            <li class="nav-item active">
                <a class="nav-link" href="home">Home</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="about">About</a>
            </li>
        </ul>
    </div>
</nav>
```

Componentes

En un componente, podemos definir propiedades que utilizaremos para mostrar información en las vistas o para condicionar el aspecto de las mismas. Para indicar al **DOM** que una de sus propiedades está ligada a una propiedad de nuestro componente, usaremos los corchetes **[ ]** , lo que denominaremos **binding de propiedades (property binding)**.

Podemos también hacer un **binding** con las clases **CSS** para condicionar un estilo al valor de una propiedad.

1. En primer lugar, nos ubicaremos en nuestra carpeta de ejercicios y crearemos el proyecto **cmpProp** mediante el comando **ng new**:

A continuación, abrimos nuestro proyecto con nuestro editor y modificamos el archivo **styles.css** añadiendo una definición para las clases **fondo** (para **div**), **td** (para celdas de la tabla que incluiremos) y **tamB** para el tamaño de los botones:

```
.fondo { background: yellow; }
td { text-align: center;
vertical-align: middle; }
.tamB { height:60px; width:200px }
```

Seguidamente, modificaremos nuestro archivo **app/app.component.ts** para que la clase **AppComponent** quede de la siguiente manera:

```
export class AppComponent {
title = 'app works!';
ponFondo = true;
mostrar = true;
habilitar = false;
referencia = "http://www.google.com";
}
```

A continuación, modificaremos el fichero **app.component.html** para que tenga el siguiente contenido:

```
<h1>{{title}}</h1>
<div class="container">
<table class="table table-bordered table-hover">
<tr>
<td><input type="checkbox" name="ponFondo"
[(ngModel)]="ponFondo">Fondo<br><br></td>
<td><div [class.fondo]="ponFondo">Hola</div></td>
</tr>
<tr>
<td colspan="2"><a [href]="referencia" target="_blank">{{
referencia }}</a></td>
</tr>
```

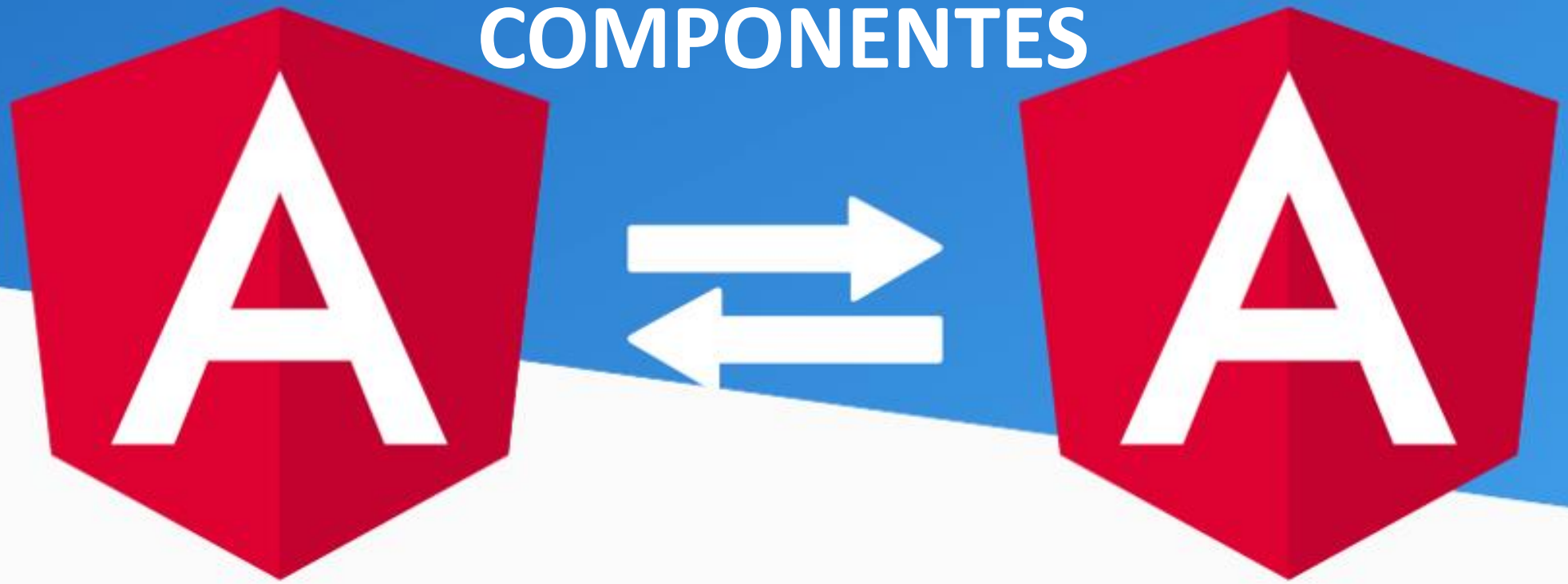
```
<tr>
<td><button class="btn btn-primary btn-success tamB"
(click)="mostrar=!mostrar"><span>{{ mostrar ? 'Mostrar' : 'Ocultar'
}}</span></button></td>
<td><div [hidden]="mostrar"><h2>Se ve</h2></div></td>
</tr>
<tr>
<td><button class="btn btn-primary tamB"
(click)="habilitar=!habilitar">{{ habilitar ? 'Habilitar' :
'Deshabilitar' }}</button></td>
<td><button [disabled]="habilitar" class="btn btn-outlineprimary
tamB">Prueba</button></td>
</tr>
</table>
</div>
```

Por último, ejecutamos la aplicación y vemos cómo se muestra en el navegador.
Para ello, nos ubicamos en nuestro el directorio de la aplicación y tecleamos **ng serve**:

Agregar el ***import { FormsModule } from '@angular/forms'*** y en el imports ***FormsModule***
En el archivo ***app.modules.ts***

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

COMUNICACIÓN ENTRE COMPONENTES



La comunicación entre componentes puede realizarse de diversas maneras (servicios, observables, etc.), pero en este ejercicio, utilizaremos los decoradores **@Input** y **@Output**

mediante **@Input**, podemos pasar datos desde un componente **padre** hacia un componente **hijo**. Con **@Output**, realizaremos lo contrario. Pasaremos datos desde un **hijo** hacia su **padre**.

Paso de datos de padres a hijos

El paso de información de padres a hijos se realiza por medio de propiedades del componente.

La información se bindea desde el template, usando los atributos de la etiqueta del componente.

Para ello, las propiedades del componente se deben decorar con ***@Input***, de modo que Angular sea capaz de saber que éstas son capaces de inicializarse, o modificarse, desde fuera.

Paso de datos de los hijos a los padres

También puede surgir la necesidad de que los hijos comuniquen a los padres cambios en la información que ellos manejan.

En estos casos Angular utiliza eventos. Es decir, cuando el hijo tiene un dato y quiere hacerlo llegar al padre, genera un evento que puede ser capturado en el padre para realizar aquellas acciones que sean necesarias.

Para definir los eventos que van de los hijos a los padres, Angular usa el decorador ***@Output***, que veremos con detalle en el siguiente artículo.

Paso de datos desde componentes padre a componentes hijo

Ahora vamos a ver cómo se realiza el envío de datos del padre al hijo, comenzando por la definición del dato en el template.

Vamos a suponer que tenemos un componente que se llama ***VisualizarCliente***. Este componente muestra la ficha de un cliente, con sus datos principales (nombre, CIF...). Para funcionar, este componente debe recibir del padre los datos del cliente que debe mostrar.

Por otra parte, tenemos otro componente que usa a ***VisualizarCliente***, que le tiene que pasar los datos que debe mostrar. Al usar el componente, tiene que pasar los datos por medio del template, usando el binding de propiedades.

El uso del componente sería como este:

```
<dw-visualizar-cliente
[nombre]="cliente.nombre"
[cif]="cliente.cif"
[direccion]="cliente.direccion" ></dw-visualizar-cliente>
```

A la vista de este código se entiende que el componente padre tendrá un objeto cliente.
El componente hijo tendrá tres propiedades, "nombre", "cif" y "direccion".
A la propiedad nombre le pasaremos el valor de "cliente.nombre", a la propiedad "cif" le pasaremos el valor de "cliente.cif", etc

Nota: nada impide haber pasado el objeto entero, de una sola vez en una única propiedad, en lugar de sus valores por separado como en el ejemplo. Sería incluso más claro y hasta preferible en muchos casos en una aplicación. Sin embargo, a fines didácticos he preferido separar los datos en varias propiedades, por el motivo de poder expresar con más claridad qué pertenece a cada componente, entre hijo y padre.

Declaración de propiedades de entrada con @Input

A continuación, vamos a ver cómo definir en el componente hijo las propiedades que son capaces de cargar datos desde el padre, Para ello usaremos el decorador ***@Input***.

(no olvidar Importar el decorador **@Input)**

```
import { Component, OnInit, Input } from '@angular/core';
```

Declarar las propiedades con su decorador *@Input*

También en el código TypeScript del componente hijo, tenemos que declarar las propiedades en la clase, igual que veníamos haciendo, solo que agregaremos el decorador @Input para indicar que esta propiedad se puede manipular desde el padre.

De momento vamos a dejar el decorador vacío, que será suficiente en la mayoría de los casos, aunque podríamos indicar algunas cuestiones como el nombre con el que conocerá la propiedad desde fuera.

```
@Input() nombre: string;
```

Si se desea, es posible aplicar un valor predeterminado a la propiedad, asignando un valor cualquiera. En ese caso, si no se entrega ese valor al usar el componente, se asumirá el definido de manera predeterminada.

```
@Input() nombre = 'Un nombre';
```

Nota: En este caso no es necesario que indiques que el tipo de la propiedad es "string", ya que el compilador de TypeScript es capaz de inferirlo en función del dato asignado.

Para realizar este ejemplo, vamos a utilizar el ejercicio de la clase 4 “Sistema de Routing” y vamos a realizar lo siguiente:

1. Generar un componente llamado ***visualizar-cliente***
2. No olvidar agregar el componente al array de declaraciones en ***app.module.ts***
3. Agregar el ***import*** del enrutado en ***app-routing.module.ts***
4. De igual forma agregar la ruta en el ***navbar***

Código completo del componente

```
import { Component, OnInit, Input } from '@angular/core’;

@Component({
   selector: 'dw-visualizar-cliente’,
   templateUrl: './visualizar-cliente.component.html’,
   styleUrls: ['./visualizar-cliente.component.css’]
})

export class VisualizarClienteComponent implements OnInit {
@Input() nombre = ‘Un nombre’;
@Input() cif = 01234 ;
@Input() dirección = ‘Una dirección’ ;

constructor() { }

ngOnInit() { }

}
```

Código completo del template

```
<p>
         Nombre: {{nombre}} <br>
         Cif: {{cif}} <br>
         direccion: {{direccion}}
</p>
```

Binding de una dirección

Estos datos, tal como se ha dicho, viajan del padre al hijo.

Si el padre modifica el valor enviado a las propiedades, también se modificará en el hijo.

Lo puedes probar si en el template donde has usado el componente pones un botón para que modifique una de sus propiedades.

```
<button (click)="cif='ESB9922'">Cambiar CIF</button>
```

Al pulsar el botón se modificará el CIF y por tanto cambiará el componente que consumía este dato.

Recuerda que el binding es de una sola dirección, como todos los binding de propiedad expresados con los corchetes.

Por tanto, si el hijo modifica la información de la propiedad, el nuevo valor no viajará hacia el padre.

Actividad: se propone una actividad muy simple, en el que un componente padre se encarga de pasar un valor al hijo para que este se muestre en su vista.

1. Generar un componente llamado ***parentcomponent***
2. Generar un componente llamado ***childcomponent***
3. No olvidar agregar el componente al array de declaraciones en ***app.module.ts***
4. Agregar el ***import*** del enrutado en ***app-routing.module.ts***
5. De igual forma agregar la ruta en el ***navbar***
6. Preparamos el componente padre según el siguiente código
7. Y el componente hijo con el código mostrado.

```
import { ParentcomponentComponent } from './components/parentcomponent/parentcomponent.component';
import { ChildcomponentComponent } from './components/childcomponent/childcomponent.component';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    AboutComponent,
    NavbarComponent,
    ParentcomponentComponent,
    ChildcomponentComponent
  ],
(app.module.ts)
```

```
import { ParentcomponentComponent } from
'./components/parentcomponent/parentcomponent.component';
import { ChildcomponentComponent } from
'./components/childcomponent/childcomponent.component';

const routes: Routes = [
  {path:'home', component:HomeComponent},
  {path:'about', component:AboutComponent},
  {path:'parent', component:ParentcomponentComponent},
  {path:'child', component:ChildcomponentComponent},
  {path: '', pathMatch: 'full', redirectTo: 'home' },
  {path: '**', pathMatch: 'full', redirectTo: 'home' },
];

(app-routing.module.ts)
```

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-parentcomponent',
  templateUrl: './parentcomponent.component.html',
  styleUrls: ['./parentcomponent.component.css']
})

export class ParentcomponentComponent implements OnInit {
nombreparent = "un nombre"
  constructor() { }

  ngOnInit(): void {
  }

}
(parentcomponent.component.ts)
```

```
<app-childcomponent [nombre]="nombreparent"></app-childcomponent>
(parentcomponent.components.html)
```

```
import { Component, OnInit, Input } from '@angular/core';

@Component({
  selector: 'app-childcomponent',
  templateUrl: './childcomponent.component.html',
  styleUrls: ['./childcomponent.component.css']
})

export class ChildcomponentComponent implements OnInit {

  @Input() nombre="" ;


  constructor() { }

  ngOnInit(): void {

  }

}
```
(childcomponent.component.ts)

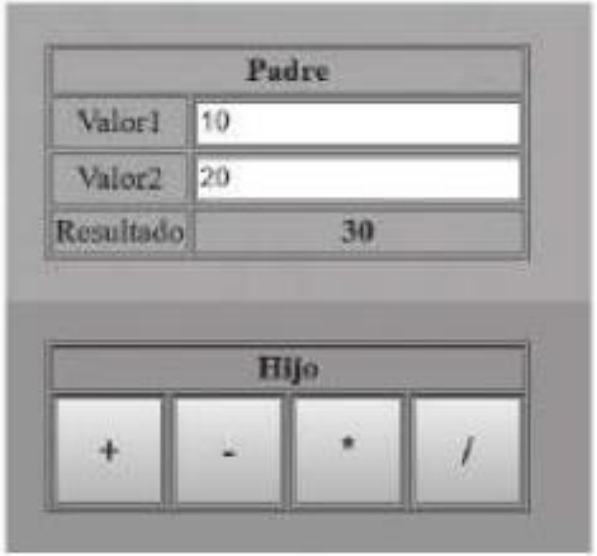
```
<p>Componente hijo</p>
<div id="hijo">
    Nombre: {{nombre}} <br>

</div>
```
(childcomponent.component.html)

Actividad: se propone un proyecto, en el que un componente padre se encarga de recoger dos valores que le pasa al hijo para realizar una operación sencilla (**suma**, **resta**, **multiplicación** y **división**) y, una vez obtenido el resultado, el mismo se devuelve al padre para que este se muestre en su vista.

Para esto vamos a utilizar el ejercicio de la clase 4 “Sistema de Routing” y vamos a realizar lo siguiente:

1. Generar un componente llamado ***hijo***
2. No olvidar agregar el componente al array de declaraciones en ***app.module.ts***
3. Agregar el ***import*** del enrutado en ***app-routing.module.ts***
4. De igual forma agregar la ruta en el ***navbar***
5. Preparamos el componente padre con el código para representar la calculadora
6. Y el componente hijo para realizar la operación al hacer click (dependiendo de si es suma, resta, multiplicación o división)

Sería algo así:



Ciclo de vida

Las directivas y componentes poseen un ciclo de vida de tal manera que se crean, se actualizan y finalmente se destruyen. Angular ofrece los **lifecycle hooks** (o “ganchos” del ciclo de vida) para que los desarrolladores puedan aprovecharlos por mediación de interfaces cuyo nombre es idéntico al evento producido, pero anteponiéndole el prefijo **ng**. Las directivas poseen un ciclo más corto.

Los **lifecycle hooks** son los siguientes:

• **ngOnChanges:** cuando se detectan cambios en las propiedades de entrada. El método recibe un objeto **SimpleChanges** de los valores de propiedad actuales y anteriores.

• **ngOnInit:** tras la primera ejecución de **ngOnChanges**. Se llama solo una vez e inicializa el componente o directiva. Usamos este método para realizar inicializaciones justo después de la construcción del componente. Es un buen lugar para cargar datos.

• **ngDoCheck:** se usa para actuar sobre los cambios que Angular no captura y que pueden tener cierto interés más allá de un simple cambio de estado.

• **ngAfterContentInit:** después de inicializar el contenido del componente. Solo existe en los componentes.

• **ngAfterContentChecked:** tras cada comprobación del contenido del componente. Solo existe en los componentes. Comprueba el contenido visualizado en el componente.

• **ngAfterViewInit:** después de que las vistas del componente se inicialicen y después de **ngAfterContentChecked**.

•**ngAfterViewChecked:** después de cada comprobación de la vista de un componente. Se llama después de **ngAfterViewInit** y de cada **ngAfterContentChecked**.

• **ngOnDestroy:** antes de destruir el componente. Usamos este método para realizar la limpieza de recursos que Angular no realiza automáticamente y es un buen lugar para avisar a otros componentes de que el componente en curso se va a destruir.

Las directivas permiten añadir a HTML un comportamiento dinámico mediante una etiqueta o un selector. Dependiendo del tipo de directiva, podemos cambiar la apariencia o comportamiento de un elemento o incluso añadir o eliminar elementos del **DOM** (***document object model*** o **modelo de objetos del documento**).

Básicamente, existen dos tipos de directivas:

- **Directivas estructurales:** alteran la estructura del DOM añadiendo, sustituyendo o eliminando elementos. Empiezan por asterisco (*).
- **ngFor:** permite iterar sobre una lista de elementos y realizar diversas acciones en un HTML como, por ejemplo, fabricar listas.
- **ngIf:** en base a una evaluación, crea o elimina elementos en el **DOM**.
- **ngSwitch:** gestiona conjuntos de tags eliminando los que no cumplan una condición.
- **Directivas de atributo:** modifican la apariencia de un elemento o modifican su comportamiento.
- **ngClass**: permite añadir o eliminar dinámicamente una o varias clases CSS en un elemento.
- **ngModel**: habilita un mecanismo de **binding bi-direccional**.
- **ngStyle**: permite asignar varios estilos **inline** a un elemento.

styles.css

```
.fa { background-color: rgb(255,246,51); }
.lg { font-size: 200%; }
```

app.component.ts

```
export class AppComponent {
title = '023 dirNgClass';
fondoAmarillo=false;
letraGrande=false;
checkFondo(){
this.fondoAmarillo=!this.fondoAmarillo; }

checkLetra(){
this.letraGrande=!this.letraGrande; }

asignaClases(){
let classes =    {
fa: this.fondoAmarillo,
lg: this.letraGrande
                 };
return classes;
               }
}
```

app.component.html

```
<h1>{{title}}</h1>
<input type="checkbox" (change)="checkFondo()">
Fondo. (fondoAmarillo = {{fondoAmarillo}})<br>
<input type="checkbox" (change)="checkLetra()">
Tam.letra. (letraGrande = {{letraGrande}})<hr>
<div [ngClass]="asignaClases()">
Este texto variará según el resultado del método.<br>
Si fondoAmarillo=true -> el fondo será amarillo.<br>
Si letraGrande=true -> las letras serán grandes.
</div>
<hr>
```

Ejemplo *ngClass*

Tal y como habíamos comentado en el ejercicio anterior, ***ngIf** permite añadir o eliminar un elemento al **DOM** si se cumple una determinada condición. A diferencia de la propiedad **visibility** de **CSS** donde se puede ocultar un elemento, **ngIf** lo elimina físicamente si la condición no se cumple, de forma que no consume recursos.

```
.uno{ border: 3px solid #F00; background-color:
lightblue;}
.dos{ border: medium double #369; background-color:
yellow; }
.tamB{ height:40px; width:300px; }
```

```
export class AppComponent {
title = '024 dirNgIf';
grupo1=false;
grupo2=false;
subgrupo11=false;
subgrupo12=false;
subgrupo21=false;
subgrupo22=false;
}
```

Ejemplo *ngIf

```
<h1>{{title}}</h1>
<button type="button" class="btn btn-success tamB" (click)="grupo1=!grupo1">
Grupo1. (grupo1 = {{grupo1}})</button><br>

<div *ngIf="grupo1" class="uno">Grupo1.<br>
<button type="button" class="btn btn-primary tamB" (click)="subgrupo11=!subgrupo11">Subgrupo11. (subgrupo11
= {{subgrupo11}})</button><br>

<div *ngIf="subgrupo11" class="dos">Subgrupo11<br></div>
<button type="button" class="btn btn-primary tamB" (click)="subgrupo12=!subgrupo12">
Subgrupo12. (subgrupo12 = {{subgrupo12}})</button><br>

<div *ngIf="subgrupo12" class="dos">Subgrupo12<br></div>
</div>
<br>
<button type="button" class="btn btn-success tamB" (click)="grupo2=!grupo2">
Grupo2. (grupo2 = {{grupo2}})</button><br>

<div *ngIf="grupo2" class="uno">Grupo2.<br>
<button type="button" class="btn btn-primary tamB" (click)="subgrupo21=!subgrupo21">
Subgrupo21. (subgrupo21 = {{subgrupo21}})</button><br>

<div *ngIf="subgrupo21" class="dos">Subgrupo21<br></div>
<button type="button" class="btn btn-primary tamB" (click)="subgrupo22=!subgrupo22">
Subgrupo22. (subgrupo22 = {{subgrupo22}})</button><br>

<div *ngIf="subgrupo22" class="dos">Subgrupo22<br></div>
</div>
```

La directiva permite insertar bloques de código de **ngFor HTML** forma dinámica, basándose en una lista de elementos (en general, arrays). La sintaxis general es la siguiente:

```
<tag *ngFor = ”let elemento of elementos” ></tag>
```

donde tag es el elemento HTML que queremos crear según el número de iteraciones (p. ej., div, p, li, etc.) y elementos en la lista de elementos que iremos asignando uno a uno a elemento mediante let.

Podemos averiguar el número del elemento que se está tratando si añadimos let i=index, tal y como se muestra a continuación:

```
<tag *ngFor = ”let elemento of elementos ; let i=index “></tag>
```

Ejemplo *ngFor

app.component.ts

```
export class AppComponent {
title = '025 dirNgFor';
public nombres = [
{nom:"uno",edad:10},
{nom:"dos",edad:20},
{nom:"tres",edad:30}
];
}
```

app.component.html

```
<h1>
{{title}}
</h1>
<div class="container"><br>
<ul class="list-group">
<li class="list-group-item list-group-item-success"
*ngFor="let nombre of nombres; let i=index">
<p>{{i + 1}} - </p>
<p>Nombre: <b>{{ nombre.nom }}</b></p>
<p>, edad: <b>{{ nombre.edad}}</b></p>
</li>
</ul>
</div>
```

ngModel es un mecanismo que permite actualizar los valores de las variables usadas en el componente y en el archivo HTML de forma bidireccional lo que se conoce como **DATABINDING** (**One Way** y **Two Way**).

*Antes de poder utilizar **ngModel** hemos de importar el **FormsModule** y añadirlo en la lista de módulos importados en **app.module.ts**

```
import { FormsModule } from '@angular/forms';
…
imports: [ BrowserModule, FormsModule ],
```

El ejemplo más típico utiliza el elemento **INPUT** para mostrar su funcionamiento y su sintaxis general es la

```
<input [(ngModel)]="variable">
```

A la notación **[( )]** también se le conoce como **banana in a box** por la forma gráfica resultante al imaginar que los corchetes representan una caja y los paréntesis una banana.

Por ejemplo, tenemos un componente llamado "cuadrado", que declara una propiedad llamada "lado". La class del componente podría quedar así:

```
export class CuadradoComponent { lado = 4; }
```

Si queremos que ese valor "lado" se vuelque dentro de un campo INPUT de formulario, tendríamos que usar algo como esto.

```
<input type="number" [ngModel]="lado">
```

para especificar el doble binding, se debe usar la sintaxis "banana in a box".

```
<input type="number" [(ngModel)]="lado">
```

Ahora, lo que se escriba en el campo INPUT también viajará hacia el modelo, actualizando el valor de la propiedad "lado" del componente

Evento ***ngModelChange***
Si lo deseas, también tienes disponible un evento llamado "***ngModelChange***" que se ejecuta cuando cambia el valor en la propiedad asociada a un ***ngModel***, con el que podríamos conseguir un comportamiento idéntico al visto en el punto anterior del doble binding, pero sin usar el binding doble.

Tienes que trabajar con ***ngModelChange*** en conjunto con la directiva ***ngModel***. Mediante ***ngModel*** asocias la propiedad que quieres asociar y entonces tendrás la posibilidad de asociar un manejador de evento a ***ngModelChange***, cada vez que esa propiedad cambia.

Dentro del manejador de evento podemos además usar una variable llamada ***$event***, en la que recibimos el nuevo valor escrito, que podríamos volcarla de nuevo a la propiedad por medio de una asignación, para conseguir el mismo efecto del binding en las dos direcciones.

```
<input type="number" [ngModel]="lado" (ngModelChange)="lado = $event">
```

app.component.html

```
<p> Tamaño del lado {{lado}} </p>
<p> 1 way binding <input type="number" [ngModel]="lado"> </p>
 <p> 2 way binding: <input type="number" [(ngModel)]="lado"> </p>
<p> Evento ngModelChange: <input type="number" [ngModel]="lado" (ngModelChange)="cambiaLado($event)"> </p>
```

app.component.ts

```
   import { Component } from '@angular/core’;

   @Component({
   selector: 'dw-cuadrado’,
   templateUrl: './cuadrado.component.html’,
   styleUrls: ['./cuadrado.component.css'] })

   export class CuadradoComponent {

         lado = 1;

         cambiaLado(valor) { this.lado = valor; }
}
```

Ejemplo *ngModel*

styles.css

```
.bg { background-color: #F9E79F; }
.bg2 { background-color: #D5F5E3; }
```

app.component.ts

```
export class AppComponent {
title = '027 dirNgModel';
nombre = '';
nombre2 = '';
cambiaNombre() {
this.nombre=this.nombre.toLowerCase();
this.nombre2=this.nombre.toUpperCase();
}
cambiaNombre2() {
this.nombre2=this.nombre2.toUpperCase();
this.nombre=this.nombre2.toLowerCase();
}
}
```

app.component.html

```
<h1>{{title}}</h1>
<br>
<div class="container bg">
<br>
<input class="form-control" [(ngModel)]="nombre"
(keyup)="cambiaNombre($event)"><br>
<input class="form-control" [(ngModel)]="nombre2"
(keyup)="cambiaNombre2($event)"><br>
</div>
<br>
<div *ngIf="nombre" class="container bg2">
<h3>Hola {{ nombre }} / {{ nombre2 }}</h3>
</div>
```

La directiva **ngSwitch** funciona como el clásico **switch** que podemos encontrar en multitud de lenguajes (p. ej., **JavaScript**).

Es decir, evalúa una expresión y compara su resultado con una serie de valores añadiendo al **DOM** el bloque asociado en caso de coincidencia. Funciona con los siguientes elementos:

Keywords	Comentarios
[ngSwitch]=”variable”	Expresión que se define como un atributo y que evalúa el contenido de variable. Utiliza la propiedad **binding**. Como variable, también puede usarse una interpolación (**{{ exp }}** como veremos en ejercicios posteriores).
***ngSwitchCase**	Define la comparación que se va a realizar. Si se cumple, se muestra el elemento HTML asociado (div, p, li, etc.).
***ngSwitchDefault**	Define el elemento por defecto que se va a mostrar cuando no hay ninguna coincidencia.

```
<elemento [ngSwitch]=”variable”>
<elemento *ngSwitchCase=”’valor1’”>…</elemento>
<elemento *ngSwitchCase=”’valor2’”>…</elemento>
<elemento *ngSwitchDefault>…</elemento>
</elemento>
```

También podríamos utilizar la siguiente sintaxis:

```
<elemento ngSwitch=”{{ variable }}”>
```

styles.css

```
.bg { background-color: #F9E79F; }
.bg2 { background-color: #D5F5E3; }
```

app.component.ts

```
opcion = "";
```

app.component.html

```
<h1>{{title}}</h1>
<br>
<div class="container bg" [ngSwitch]="opcion"> Valor seleccionado:
<div class="container text-success" *ngSwitchCase="'A'">Ha elegido la
opcion: <b>A</b></div>
<div class="container text-success" *ngSwitchCase="'B'">Ha elegido la
opcion: <b>B</b></div>
<div class="container text-success" *ngSwitchCase="'C'">Ha elegido la
opcion: <b>C</b></div>
<div class="container text-danger" *ngSwitchDefault><b>Sin selección</b></div>
</div>
<br>
<div class="container">
<button class="btn btn-primary btn-lg" (click)="opcion='A'">A</button>
<button class="btn btn-primary btn-md" (click)="opcion='B'">B</button>
<button class="btn btn-primary btn-sm" (click)="opcion='C'">C</button>
</div>
<br>
<div class="container bg2"><h5>Boton pulsado : {{opcion}}</h5></div>
```

Blog
Front-end
CSS
DEV
Transiciones y animaciones CSS.
Desarrollo de software
height: .1%;
opacity: 1;
}
to {
width:

Transiciones y Animaciones en CSS

Las transiciones CSS nos permiten pasar de un estado ***A*** a un estado ***B*** a partir de un valor inicial hasta un valor final de manera gradual, siendo el navegador quien calcula los pasos intermedios y sin que nosotros podamos interferir en ellos.

Las animaciones nos permiten, al igual que las transiciones, pasar un elemento del estado ***X*** al estado ***Z***, permitiendo además definir un número variable de estados intermedios. Es decir, nos ofrecen un mayor control sobre el cambio de estado y las fases que atraviesa este.

Estos últimos utilizan reglas ***@keyframe*** nos permite definir los fotogramas clave que tendrá la animación mediante bloques definidos por un valor en %

Un error común es tratar de animar una propiedad display con valor ***none*** a un valor block (o el que se necesite). Para saber si una propiedad CSS se puede ***transicionar/animar*** es determinar <u>si existe esa propiedad con un valor intermedio</u>: de no mostrarse (display: ***none***) a mostrarse (display: ***block***) no hay un valor "schrödinger-block" intermedio, por lo tanto, esto nos indica que la propiedad display no es ***animable/transicionable***.

Propiedades CSS a las que se les pueden aplicar animaciones y/o transiciones:

•Propiedades de color: color, opacity

•Propiedades de columnas: column-width, column-count, column-gap, column-rule-color, column-rule-width.

•Propiedades de texto: letter-spacing, tab-size, text-indent, word-spacing.

•Propiedades de decoración de texto: text-decoration-color, text-shadow.

•Propiedades flex: flex-basis, flex-grow, flex-shrink, order.

•Propiedades de fondo y bordes: background-color, background-position, background-size, border-witdh, border-radius, border-image.

•Propiedades del box-model: box-shadow, margin, padding, max-height, min-height, height, max-width, min-width, width, visibility, vertical-align.

•Propiedades de posicionamiento: bottom, left, right, top, z-index.

•Propiedades de tipografía: font-weight, font-strech, font-size, line-height, font-size-adjust.

•Propiedades object-fit: object-position.

•Propiedades de interacción: outline-color, outline-width, outline-offset.

•Propiedades de transformación: transform, transform-origin, perspective, perspective-origin.

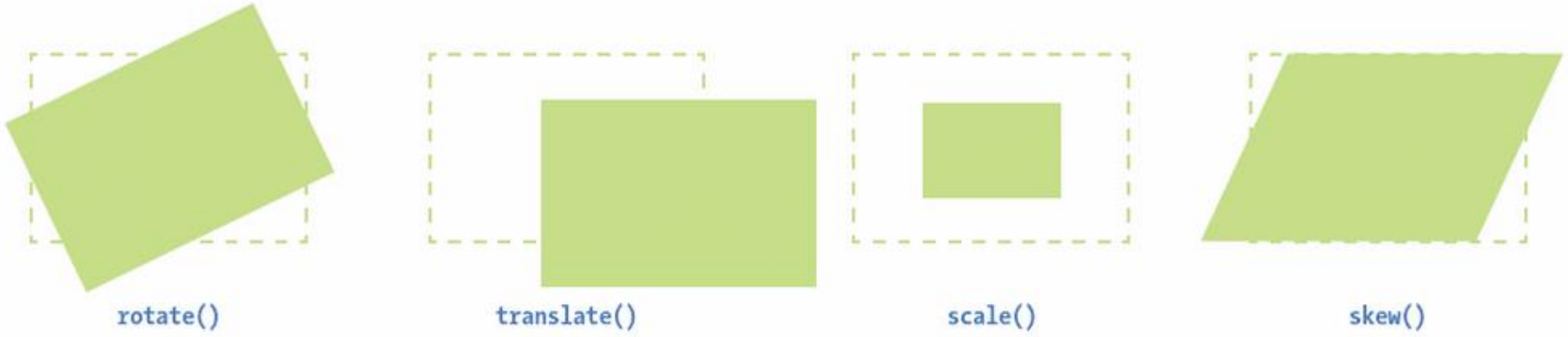
•Propiedades de forma: shape-outside, shape-margin, shape-image-threshold.

Transformaciones CSS

El módulo de transformaciones CCS3 permite rotar, reubicar, redimensionar, sesgar elementos HTML en dos y tres dimensiones, siendo estas también animables.

métodos:

- ***Rotaciones*** : rotate(ndeg), rotateZ(ndeg), rotate3d(ndeg)
- ***Translaciones***: translate(Xpx,Ypx), translateX(Xpx), translateY(Ypx)
- ***Escalar***: scale(X, Y), scaleX(X), scaleY(Y)
- ***Sesgar***: skew(Xdeg, Ydeg), skewX(Xdeg), skewY(Ydeg)



Las transiciones CSS

Proporcionan una forma de animar los cambios de las propiedades CSS, en lugar de que los cambios surtan efecto de manera instantánea. Por ejemplo, si cambias el color de un elemento de blanco a negro, normalmente el cambio es instantáneo. Al habilitar las transiciones CSS, el cambio sucede en un intervalo de tiempo que puedes especificar, siguiendo una curva de aceleración que puedes personalizar.

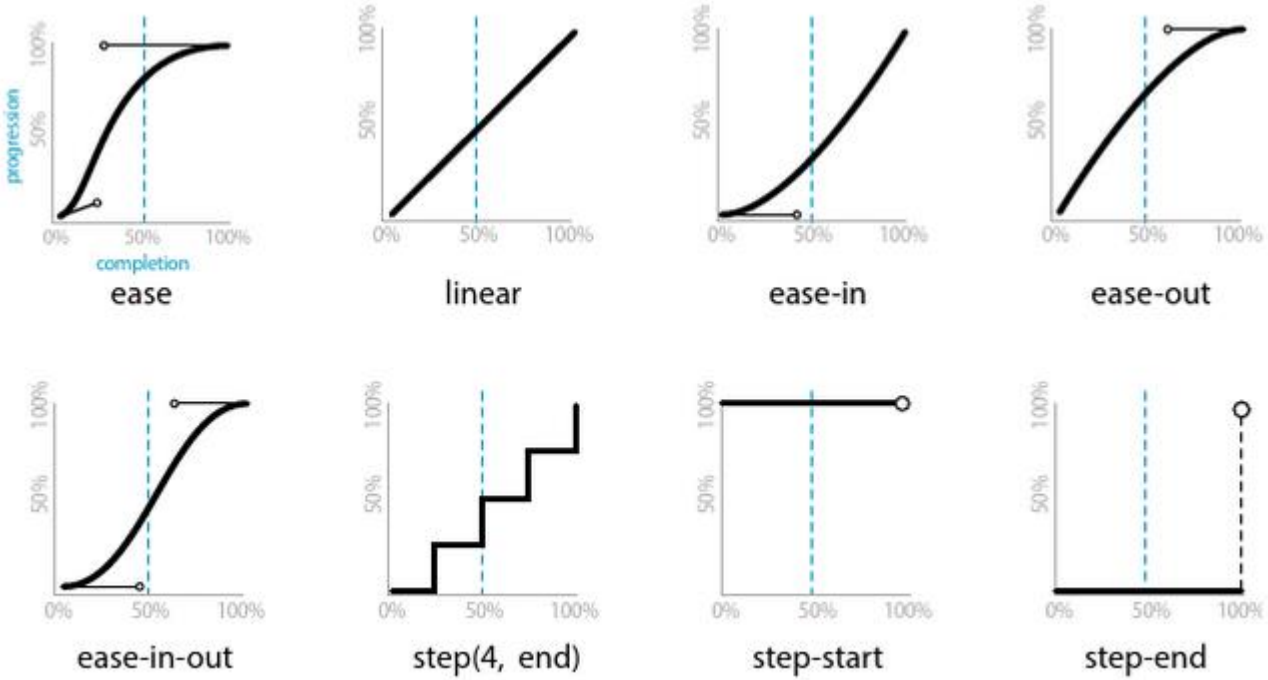
Hay 4 propiedades para describir las transiciones CSS:

1. ***transition-property***. Propiedad o lista de propiedades para animar podemos escribir all, que significa todas.
2. ***transition-duration***. Podemos especificar el tiempo que debe durar la animación en segundos s y milisegundos ms.
3. ***transition-timing-function***. La función de temporización describe como se distribuye el proceso de la animación a lo largo del tiempo. Comenzando lentamente y luego irá rápido o viceversa, esta propiedad acepta dos tipos de valores: una curva de Bézier o pasos.
4. ***transition-delay***. Retrazo antes de iniciar la animación, esta también se puede especificar en segundos y milisegundos, se pueden especificar números negativos

transition-timing-function.

Esta propiedad puede tomar los siguientes valores:

1. ease, ease-in, ease-out, ease-in-out
2. linear
3. ease
4. ***`cubic-bezier(x2, y2, x3, y3)`***.
5. step, step-start, step-end.



Así como la curva Bezier es una función de temporización, la función steps() indica el número de pasos en los que la animación se debe dividir, acepta dos parámetros el primero es un número entero y el segundo es opcional y puede ser ***start*** o ***end***, *dictando si el cambio de estado debe tener lugar al principio o al final de un paso*.

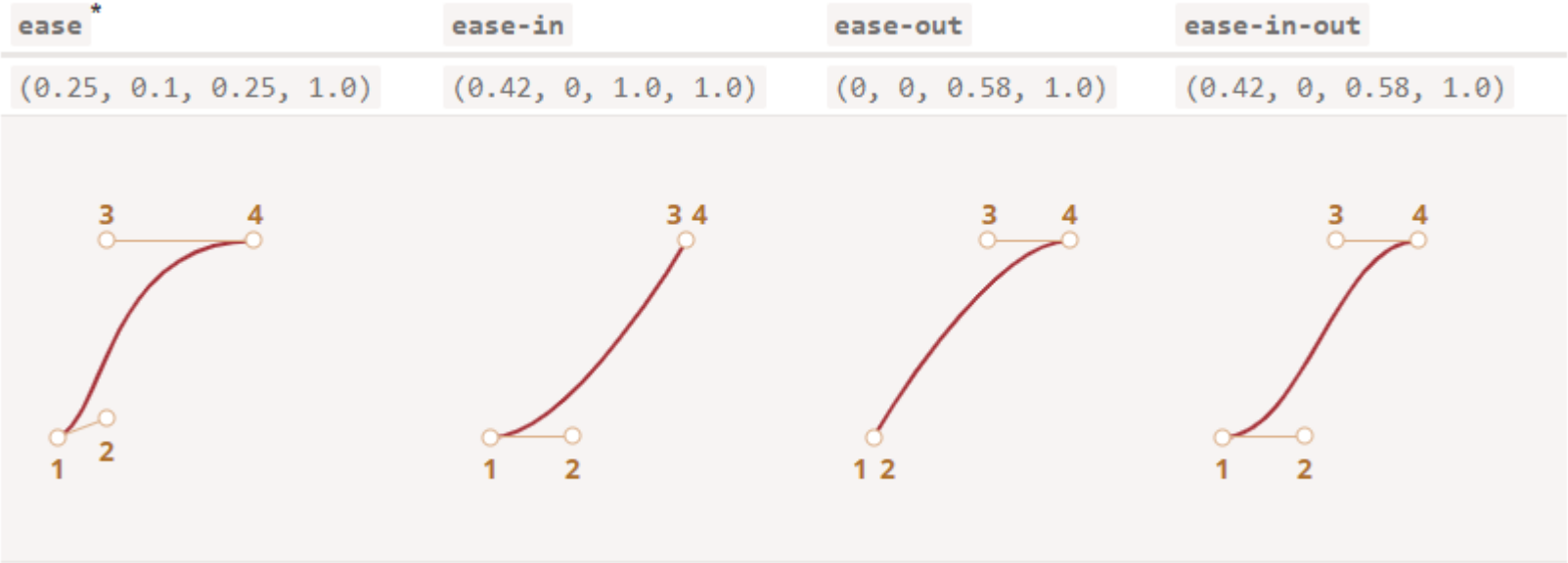
```
<!Doctype html>
<header>
<style>
div{
                width: 67px;
                height: 66px;
                background: url('animacion.png') left;
                transition: 1s steps(12) ;

                }

                div:hover{
                background-position: -800px 0;
                }
</style>
</header>
<body>
<div></div>
</body>
</html>
```

Curva de Bézier

La función de temporización se puede establecer como una curva de Bézier con 4 puntos de control que satisfacen las condiciones:

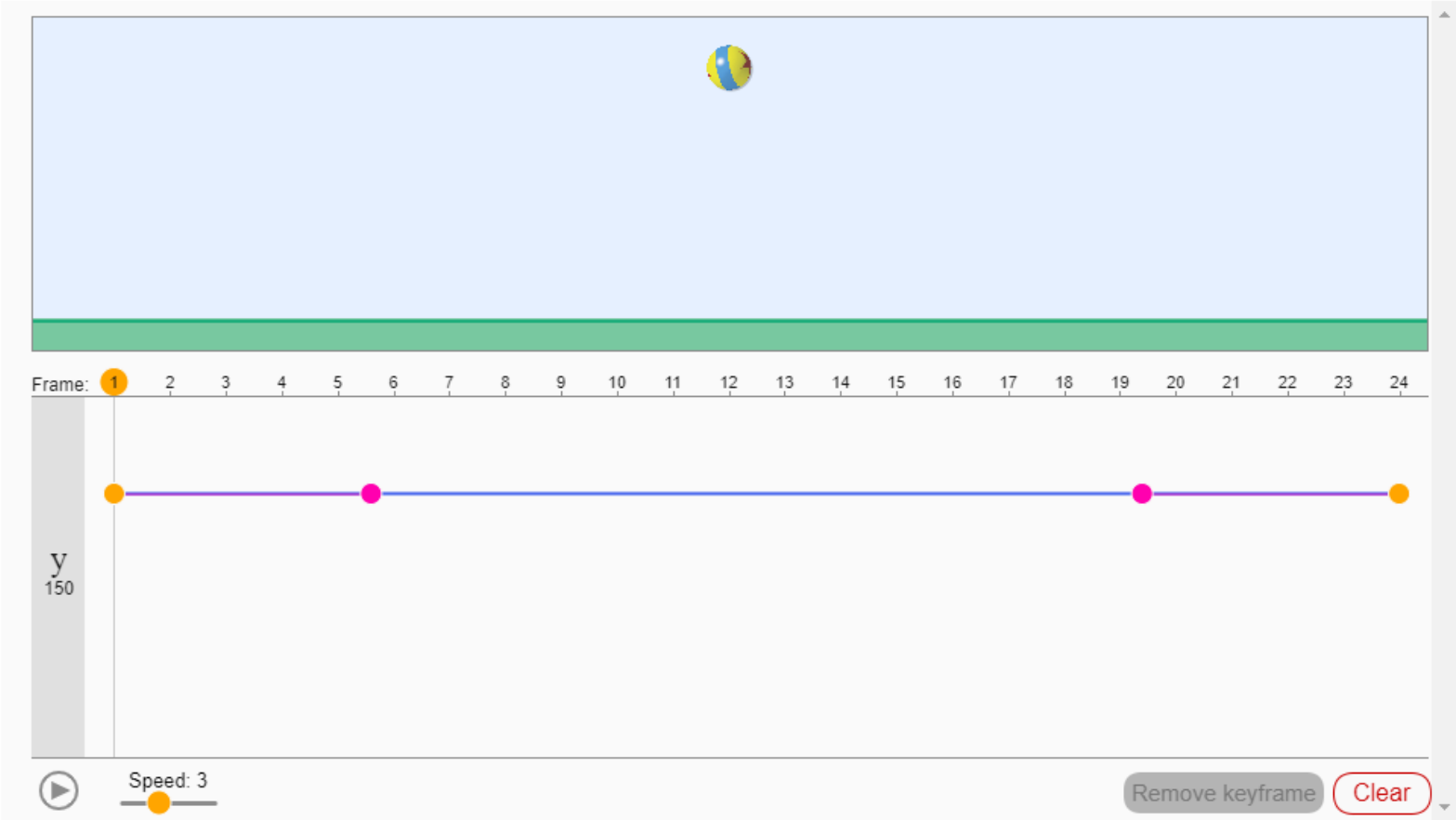
1. Primer punto de control: `(0,0)`.
2. Último punto de control: `(1,1)`.
3. Para los puntos intermedios, los valores de x deben estar en el intervalo `0..1`, y puede ser cualquier cosa.
4. La sintaxis de una curva de Bézier en CSS: ***`cubic-bezier(x2, y2, x3, y3)`***. Aquí necesitamos especificar solo los puntos de control segundo y tercero, porque el primero está fijado a `(0,0)` y el cuarto es `(1,1)`.
5. El eje ***x*** especifica el tiempo y el eje ***y*** la finalización del proceso

Otros nombres son abreviaturas para la siguiente ***`cubic-bezier`***:



Calculadora online: https://www.desmos.com/calculator/d1ofwre0fr?lang=es; https://cubic-bezier.com/

Usa el siguiente programa para crear una animación de una pelota que rebota. Ve si puedes hacer que la pelota se acelere cuando pegue en el piso.



https://es.khanacademy.org/computing/pixar/animate/ball/pi/animation-with-bezier-curves

```html
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="css/style.css">
</head>

<body>

  <img class="train" src="https://js.cx/clipart/train.gif"
        onclick="this.style.left='450px'">

</body>

</html>
```

```css
.train {
  position: relative;
  cursor: pointer;
  width: 177px;
  height: 160px;
  left: 0;
  transition: left 5s cubic-bezier(0.5, 0.5, 0.5, 1.0);
}
/*
cubic-bezier(0.0, 0.5, 0.5, 1.0)
ease
ease-in
ease-out
ease-in-out
*/
```

Fotogramas clave (Keyframes)

Podemos unir múltiples animaciones simples juntas usando la regla CSS ***`@keyframes`***, y se especifica el "***nombre***" de la animación y las reglas: qué, cuándo y dónde animar. Luego, usando la propiedad ***`animation:`***, adjuntamos la animación al elemento y especificamos parámetros adicionales para él.

- ***animation-name***. Define una lista con la secuencia de las animaciones cuando esta se relaciona con los fotogramas (keyframes)
- ***animation-duration***. Define la duración de la animación
- ***animation-timing-function***. Define como se distribuye el proceso de animación a lo largo del timpo
- ***animation-delay***. Define el retrazo de tiempo antes de comenzar la animación
- ***animation-iteration-count***. Define la cantidad de veces que se repite la animación
- ***animation-direction***. Define el sentido en que se reproduce la animación

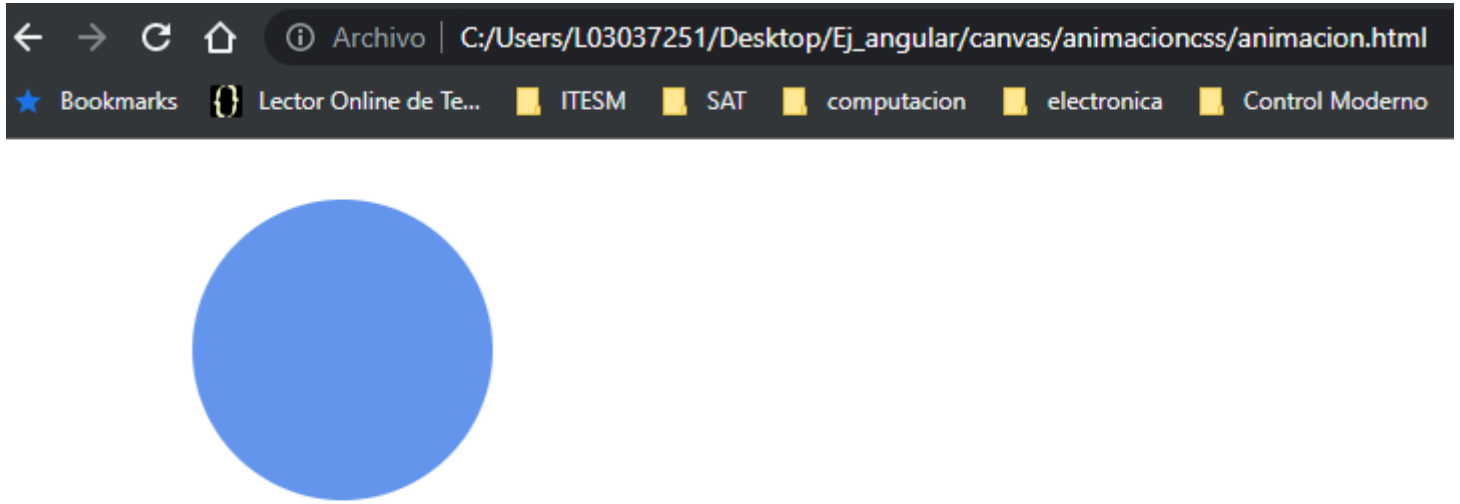
```html
<!DOCTYPE html>
<html>
    <head></head>
    <body>
        <div class="progress"></div>

        <style>
          @keyframes go-left-right {        /* dale un nombre: "go-left-right" */
            from { left: 0px; }             /* animar desde la izquierda: 0px */
            to { left: calc(100% - 50px); } /* animar a la derecha: 100%-50px */
          }

          .progress {
            animation: go-left-right 3s infinite alternate;
            /* aplicar la animación "go-left-right" al elemento
               duración 3 segundos
               número de veces: infinitas
               alternar la dirección cada vez
            */

            position: relative;
            border: 2px solid green;
            width: 50px;
            height: 20px;
            background: lime;
          }
        </style>
    </body>
</html>
```

animacion.html

```html
<!DOCTYPE html>
<html>
    <head>
        <title>animación CSS</title>
        <link rel="stylesheet" href="estilos.css" >
    </head>
    <body>
            <div class="circulo"></div>
    </body>
</html>
```

estilos.css

```css
*{
    margin:0;
    padding:0;
    box-sizing: border-box;
}

.circulo{
    margin-top:30px;
    margin-left:20px;
    width: 150px;
    height: 150px;
    background-color: cornflowerblue;
    border-radius:50%;
    /*Animacion CSS*/
    animation-name: moverCirculo;
    animation-duration: 1s;
    animation-iteration-count: infinite;
    animation-direction:alternate-reverse;
    animation-timing-function: linear;

}

@keyframes moverCirculo{
    0%{ transform: translateX(0px); }

    100%{ transform: translateX(100px) }
}
```


Archivo
C:/Users/L03037251/Desktop/Ej_angular/canvas/animacioncss/animacion.html
Bookmarks
Lector Online de Te...
ITESM
SAT
computacion
electronica
Control Moderno

CANVAS
HTML
5

Historia

Canvas fue introducido inicialmente por Apple para su uso dentro de su propio componente de Mac OS X Surgido en 2004, para empujar aplicaciones como widgets de Dashboard y el navegador Safari. Más tarde, en 2005, se adoptó en la versión 1.8 de los navegadores Gecko y Opera en 2006. Fue estandarizado por el Grupo de Trabajo de Tecnologías de Aplicación de hipertexto Web (WHATWG por sus siglas en inglés) sobre las nuevas especificaciones propuestas para tecnologías web de última generación.

Uso

El Canvas consiste en una región dibujable definida en el código HTML con atributos de *altura* y *ancho*. El código JavaScript puede acceder a la zona a través de un conjunto completo de funciones similares a las de otras APIs comunes de dibujo 2D, permitiendo así que los gráficos sean generados dinámicamente. Algunos de los usos previstos incluyen construcción de gráficos, animaciones, juegos, y la composición de imágenes.

Lo primero que tenemos que hacer para trabajar con canvas es crear un lienzo y a partir de ese lienzo incluir elementos y modificarlos para esto utilizamos la etiqueta HTML5 <canvas>

```
<canvas id="canvas" width="200" height="100"></canvas>
```

Para acceder a las funcionalidades de ***canvas*** es necesario acceder al DOM para darle formato a nuestro canvas y posteriormente añadimos el contexto con el que vamos a dibujar en nuestro lienzo, en este caso es de dos dimensiones. También es posible añadir un contexto en tres dimensiones, pero para este contexto se necesitan librerías javascript como puede ser ***three.js*** o tecnología ***webGL***.

Para ello crearemos dos variables:

```
var canvas = null;
var ctx    = null;
```

Y las inicializamos:

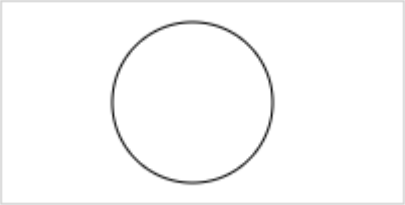
```
canvas = document.getElementById("canvas");
ctx    = canvas.getContext('2d');
```

Atributo	Descripción
canvas	Se refiere al contexto del canvas. El uso más común es para acceder a los atributos del canvas ancho y alto: context.canvas.width y contex.canvas.height respectivamente.
fillStyle	Especifica un color, gradiente o patrón que el contexto usa en subsecuencia para llenar una forma o figura.
font	Especifica la letra que el contexto usa cuando se llama fillText() o strokeText()
globalAlpha	Ajuste del Alpha global, debe ser un número entre 0 (totalmente transparente) y 1 (totalmente opaco). El navegador multiplica el valor Alpha de cada pixel por la propiedad del Alpha global, inclusive cuando se dibujan imágenes.
globalComposit-eOperation	Determina como el navegador dibuja un objeto sobre otro.
lineCap	Especifica como el navegador dibuja los puntos finales de una linea. Se puede especificar uno de los tres siguientes valores: butt, round y square. El default es butt
lineWidht	Determina el ancho en pixeles de las líneas que se dibujan en un canvas. El valor no debe ser negativo, por default es 1
lineJoin	Especifica como las líneas se unen cuando sus puntos finales se encuentran, los valores posibles son: bevel, round y miter, dafault es miter.

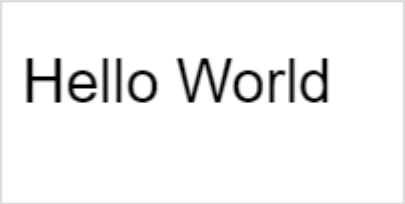
Atributo	Descripción
miterLimit	Especifica como dibujar una unión de linea miter
shadowBlur	Especifica como el navegador extiende hacia afuera una forma, su valor no es en pixeles
shadowColor	Especifica el color que el navegador utiliza para dibujar las formas
shadowOffsetX	Especifica el offset horizontal, en pixeles
shadowOffsetY	Especifica el offset vertical, en pi
strokeStyle	Especifica el estilo utilizado para el trazo, el valor puede ser un color, gradiente o patrón
textAlign	Determina la colocación horizontal del texto cuando se utiliza fillText() o strokeText()
textBaseline	Determina la colocación vertical del texto cuando se utiliza filltext() o strokeText()

Métodos	Descripción
fillRect(x, y, width, height)	Permite dibujar un rectángulo relleno
strokeRect(x, y, with, height)	Permite dibujar un contorno de un rectángulo
clearRect(x, y, width, height)	Podemos borrar cualquier elemento dibujado dentro del área definida
beginPath()	Para indicar que se realizará un dibujo
moveTo(x, y)	Indicamos la ubicación de inicio
lineTo(x, y)	Indicamos la ubicación de fin
stroke()	Para hacerla visible
arc(x, y, radio, angulo, sentido)	Se utiliza para dibujar un circulo o arco de circulo, el ángulo debe especificarse en radianes, el arco no será visible hasta no llamar a fill() o stroke().
fillText(texto, x, y)	Se utiliza para dibuar texto dentro del canvas
drawImage(obj-image, x, y)	Se utiliza para añadir una imagen, previamente es necesario crear un objeto de tipo imagen y establecer su propiedad src, ejemplo: Var img = new Image(); Img.src = ‘img/imagen.png’;

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.moveTo(0, 0);
ctx.lineTo(200, 100);
ctx.stroke();
</script>
```



```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(95, 50, 40, 0, 2 * Math.PI);
ctx.stroke();
</script>
```




```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.font = "30px Arial";
ctx.fillText("Hello World", 10, 50);
</script>
```

```
<script>
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
var img = document.getElementById("scream");
ctx.drawImage(img, 10, 10);
</script>
```

```html
<!DOCTYPE html>
<html>
    <head>
        <title>ejemplo canvas</title>
<script>
    var cnv, contexto, posX = 60;
    function animacion(){

        contexto.clearRect(0,0,600,200);
        contexto.beginPath();
        contexto.arc(posX,60,50,0,2*Math.PI,true);
        contexto.fill();
        posX += 10;

        setTimeout(animacion,50);
     }

     window.onload = function(){
        cnv = document.getElementById('canvas');
        contexto = cnv.getContext('2d');
        animacion();
     }
</script>
    </head>
    <body>
        <canvas width="600" height="200" id="canvas"
                style="background-color:rgb(136, 191,
197)"></canvas>
    </body>
</html>
```