Programación Web: HTML, CSS, JavaScript

Miguel Ortuño
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

Diciembre de 2023

Presentaciones para clase.
Material docente en abierto de la Universidad Rey Juan Carlos
para las asignaturas
Aplicaciones Telemáticas
y
Desarrollo de Aplicaciones Telemáticas

HTML

Introducción a HTML (1)

- HTML *Hypertext Markup Language* es un lenguaje de marcado, inicialmente se usa para que navegadores web compongan páginas con diverso contenido: texto, enlaces, imágenes, audio, vídeo, etc. Páginas *estáticas*: se editaban generalmente a mano
- El éxito de internet y HTML como interfaz de usuario hace que a finales de los años 1990 aparezcan tecnologías que permiten páginas web dinámicas: CGI, PHP, ASP...
- En 1995, JavaScript permite programar scripts en el navegador web, típicamente para componer código HTML
- Muy importante: no confundir HTML con HTTP

Introducción a HTML (2)

- A mediados de los años 2000 se desarrolla una nueva generación de herramientas para facilitar la programación de aplicaciones web: Ruby on Rails, Django...
- En esta misma época, año 2005, la aparición de AJAX permite que las aplicaciones web comiencen a semejarse a las aplicaciones de escritorio
- A principios de la década de 2010, la madurez de las tecnologías web hace que empiecen a usarse también fuera del navegador. *JavaScript everywhere*. Nodejs. HTML como plataforma para interfaz gráfico en el escritorio

Lenguajes de marcado

HTML es un *lenguaje de marcado*: un sistema que permite incluir metainformación en un documento, esto es, información sobre la información

- La metainformación tiene que distinguirse sintácticamente del texto.
- Es la evolución del *lapiz azul* con el que tradicionalmente se editaban documentos cuando la tecnología era analógica
  - Ejemplo de lenguaje de marcado muy elemental: redacto un documento en un procesador de textos, lo imprimo y alguien lo revisa, incluyendo anotaciones a mano. Las anotaciones (metainformación) se distingue fácilmente del texto original Para hacer algo semejante de forma digital, es necesaria una sintaxis que separe el texto de la metainformación
- Ejemplos de lenguajes de marcado: troff, LaTeX, JsonML, SGML, HTML, XML

XML

- XML *Extensible Markup Language*
  Es una forma de describir datos jerárquicamente. Estándar para transferir información entre distintos sistemas sin tener que adaptarlos a cada plataforma concreta, y de forma que sea fácil de leer por un humano y fácil de procesar por un ordenador
- Creado en 1996 por el W3C (*World Wide Web Consortium*)
  Dos versiones: XML 1.0 y XML 1.1
- Algunos autores lo consideran un lenguaje de marcado, otros, un metalenguaje de marcado
- Proviene de SGML, *Standard Generalized Markup Language*, norma ISO 8879:1986
  SGML es un metalenguaje, un lenguaje para definir lenguajes de marcado,

- XML tiene una sintaxis muy similar a la de HTML porque ambos provienen de SGML
- XML y HTML no son lenguajes alternativos
  - XML está diseñado para describir y comunicar datos datos de máquina a máquina
  - HTML está diseñado para presentar en pantalla datos con formato. De máquina a persona

En HTML, como en cualquier lenguaje de marcado, es esencial separar los aspectos semánticos del texto del formato de la representación gráfica

- Con la aparición de las plataformas móviles (smartphones y tablets), esto se vuelve aún más importante
- Las primeras versiones de HTML no eran muy rigurosas en esto, pero se ha ido corrigiendo gradualmente en cada nueva especificación

Internet moderno: HTML

- HTML es la base de la World Wide Web, y por tanto, de la internet moderna
- Para los usuarios, WWW e internet son sinónimos. Pero nosotros debemos distinguirlo

Internet antes del web:

- Los predecesores de internet aparecen en los años 1960
- TCP/IP se desarrolla durante la década de 1970
  - 1 de enero de 1983: *flag day* en que la red ARPANET migra desde NCP hasta TCP/IP
- En la internet primitiva se usaban servicios como el correo electrónico, ftp, telnet, usenet, gopher, irc y algunos otros

Versiones de HTML (1)

- Años 1989-1991. Tim Berners-Lee, un físico del CERN (*European Organization for Nuclear Research, Conseil Européen pour la Recherche Nucléaire*) crea un sistema de hipertexto para internet al que llama WWW, *World Wide Web*.
  - Para ello desarrolla el lenguaje HTML junto con el protocolo HTTP
  - Lo que en origen era un servicio más para un ámbito muy específico, tiene un éxito arrollador que cambia no solo internet y la informática, sino las comunicaciones humanas
  - El resto de servicios de internet siguen diferenciándose del WWW, pero casi todos ellos acaban teniendo un interfaz de usuario web, lo que hace que el usuario lo perciba como la misma cosa

Versiones de HTML (2)

- Año 1995. HTML 2.0. Publicado por el IETF (Internet Engineering Task Force). Añade formularios, tablas, y soporte para internacionalización, entre otros
- Año 1997. HTML 3.2. *Guerra de los navegadores*. Microsoft Internet Explorer y Netscape Navigator intentaban prevalecer en el mercado, añadiendo características propias. HTML 3.2 busca que ambos navegadores vuelvan a ser compatibles. Añade características muy desacertadas, como la etiqueta *font* y el atributo *color*

Versiones de HTML (3)

- Versión 4.0. Año 1997
  Normaliza el uso de marcos *frames*, disponible desde Netscape Navigator 2 (año 1995)
  Los marcos eran documentos HTML dentro de documentos HTML. Concepto problemático, han ido desapareciendo
  Tres variantes de HTML 4.0
  - *Strict*, donde se prohiben elementos obsoletos
  - *Transitional*, que admite elementos obsoletos
  - *Frameset*, solo marcos
- Versión 4.1. Año 2000
  La versión más usada, hasta la aparición de HTML 5

Versiones de HTML (4)

- Año 2004. El W3C decide abandonar HTML y migrar a XHTML
  - XHTML: Extensible Hypertext Markup Language. Lenguaje muy similar a HTML, pero que sigue estrictamente la sintaxis de XML

  El desarrollo de HTML lo retoma el WHATWG (Web Hypertext Application Technology Working Group: Google Apple, Mozilla, Opera)
- Año 2008. Primer borrador de HTML 5 publicado por WHATWG
- Año 2009. El W3C abandona XHTML y vuelve a HTML5

Versiones de HTML (5)

- Año 2014. HTML 5.0
  - Define de forma precisa qué hacer con páginas incorrectas
  - Muchas mejoras en interoperabilidad
  - Mucho mejor soporte para dispositivos móviles
  - Audio y video
  - Gráficos vectoriales
  - Muchas APIs nuevas, como la geolocalización
- Año 2017. HTML 5.2

En la actualidad cualquier desarrollo debería centrarse en HTML 5

- La compatibilidad con HTML 4 es bastante buena
- Existen técnicas y herramientas que permiten que páginas HTML 5 se muestren correctamente en navegadores antiguos

Toda la información contenida en las transparencias de esta asignatura es válida en HTML 4 y HTML 5, salvo indicación contraria

Adobe Flash

Otra de las grandes ventajas de HTML 5 es que permite prescindir de Flash

- Adobe Flash es una plataforma software desarrollada por Adobe Systems para mostrar animaciones, gráficos vectoriales, vídeos, audio, contenido interactivo...
- Muy popular entre los años 2000 y 2010
- Muy problemático. Ya en el año 2000 se publican artículos como *Flash: 99 % Bad*, (J.Nielsen)
  No estándar. Dependencia del fabricante. Anima a desarrollar contenido centrado en la apariencia gráfica externa, no en la usabilidad y la semántica
- No soportado por Apple

Anunciada su desaparición oficial para el año 2020

Sintaxis HTML: composición de un elemento

Un documento HTML está compuesto por elementos (*elements*)
Un elemento puede ir

- A continuación de otro elemento
- Dentro de otro elemento. Esto es muy habitual, los documentos típicos tienen muchos niveles anidados

La mayoría de los elementos están formados por:

- Etiqueta de apertura (*start tag*)
- Contenido
- Etiqueta de cierre (*end tag*)

Ejemplo:

```
<h1>Este elemento es un título de nivel 1</h1>
```

Una etiqueta de apertura sencilla está formada por:

- Signo de menor
- Nombre de la etiqueta
- Signo de mayor

Ejemplo:

```
<h1>
```

Una etiqueta de cierre está formada por:

- Signo de menor
- Barra (*slash*)
- Nombre de la etiqueta
- Signo de mayor

Ejemplo:

```
</h1>
```

Lo habitual y recomendable es no poner ningún espacio ni después del < ni antes del >

- Un espacio después del < es un error
  Ejemplo :
  ```
  < h1>
  ```
  Esto es un ERROR
- Un espacio antes del > es legal, pero no es recomendable
  Ejemplo :
  ```
  <h1 >
  ```

- En la mayoría de elementos la etiqueta de cierre es obligatoria Ejemplo:

```
<pre>Texto preformateado, con fuente de ancho fijo. Se mantienen
los saltos de línea y los espacios    consecutivos</pre>
```

- En algunos elementos se puede omitir la etiqueta de cierre. Aunque no es recomendable. Ejemplo :

```
<p>Esto es un párrafo correcto</p>
<p>Esto también es un párrafo correcto
```

- En algunos elementos, los de tipo *void*, no puede haber ni contenido ni etiqueta de cierre. Solo pueden tener, opcionalmente, atributos. Ejemplo :

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

Esto es un ERROR

Elementos de tipo void muy habituales son: *br*, *hr*, *meta*, *link*, *img*, *input*

- En HTML 4.01 también son de tipo void: *area*, *base*, *col*, *param*
- HTML 5 añade: *source*

¿Etiqueta h1 o elemento h1?

Como hemos visto, HTML está formado por elementos, que siempre tienen etiqueta de apertura y que pueden tener contenido y etiqueta de cierre

Consideremos por ejemplo

```
<h1>Introducción</h1>
```

- En rigor deberíamos decir *el elemento h1*, no *la etiqueta h1*
- Pero en muchos contextos es habitual y por tanto aceptable hablar de *la etiqueta h1*, se entiende que es una sinécdoque, nos estamos refiriendo a *el elemento que empieza por la etiqueta h1*

Sintaxis de HTML: distribución de los elementos

Un documento HTML está formado por

- Declaración de tipo
- Un elemento *html*, que contiene
  - Un elemento *head*, que contiene
    - Título
    - Codificación de caracteres
  - Un elemento *body*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hola mundo en HTML</title>
    <meta charset="utf-8">
  </head>
  <body>
    Hola, mundo.
  </body>
</html>
```

- Algunas etiquetas de estos elementos se pueden omitir en ciertas circunstancias y el documento sigue siendo válido, quedan sobreentendidas (head, body, html). Pero siempre es preferible ponerlo todo
- Otros elementos como el título y la codificación de caracteres son obligatorios. Si se omiten, herramientas como `https://validator.w3.org` nos indicarán que el documento es erróneo
  - A pesar de eso, los navegadores podrán mostrar el documento de forma satisfactoria, con lo que es muy habitual que nadie se preocupe de corregir estos errores
  - El problema se agrava cuando distintos navegadores tratan los errores de forma distinta

Validación del código

En esta asignatura haremos énfasis en generar siempre código *correcto*, tomando como referencia el *W3C Markup Validation Service*

- Tu código no puede dar ningún error
- Aceptaremos algunos *warnings*, otros será preferible evitarlos

Pregunta típica: *Mi página se ve bien ¿qué más da que tenga errores?*

- Respuesta: no has encontrado ningún error en los navegadores en los que has probado. Pero pueden darse en otros navegadores, en otras plataformas, en versiones del pasado y en versiones del futuro

Cuidado:

- No confundas el *W3C Markup Validation Service* con el *Nu Html Checker* del W3C. De lo contrario, tomarías como erróneas páginas correctas en XHTML o lenguajes similares

DOCTYPE

La declaración `<!DOCTYPE html>` es obligatoria al comienzo de un documento HTML 5

- En rigor no es parte de HTML (no es un elemento HTML) sino una instrucción que le dice al navegador que lo que viene a continuación es un documento HTML, versión 5

En HTML 4.x y anteriores, esto era más complicado
Ejemplos:

- ```
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
  ```

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

Case insensitive

HTML es insensible a mayúsculas (*case insensitive*), aunque lo habitual es usar siempre las minúsculas.
Hay una excepción:

```
<!DOCTYPE html>
<!doctype html>
```

Ambas formas son idénticas y correctas, pero lo habitual es usar la primera, posiblemente por influencia de XHTML (que es sensible a mayúsculas, y donde la única forma válida es la primera)

Comentarios

Los comentarios son iguales que en XML. Se pueden poner en cualquier lugar del documento

```
<!-- Esto es un comentario -->
```

Etiquetas autocerradas

En XML, cuando un elemento no tiene texto, hay dos alternativas posibles

- Usar una etiqueta de cierre y otra de apertura

  ```
  <holamundo></holamundo>
  ```

- Usar una etiqueta *autocerrada*

  ```
  <holamundo/>
  ```

  Signo de menor, nombre, barra, signo de mayor

Por influencia de XML, algunos desarrolladores o herramientas de HTML usan las etiquetas autocerradas

- En los elementos void son válidas. Pero no aportan nada, no tienen significado especial
- En otros elementos son incorrectas. Aunque el navegador suele ignorarlos y mostrar la página igualmente

Conclusión: no debemos usar etiquetas autocerradas

Atributos

Dentro de la etiqueta de apertura puede haber uno o más *atributos*, que son modificadores del elemento
Ejemplo:

```
<html lang="es-ES">
Esto es texto en español de España
```

- El atributo *lang* indica el idioma del texto, especificado en ISO 639-1
- Su sintaxis es similar pero no idéntica a la variable LANG de Unix, donde se indicaría `es_ES.UTF-8`

- Un atributo es un par formado por un nombre y un valor. Su sintaxis es
  - Nombre del atributo
  - Signo igual
  - Valor del atributo
    - Siempre es de tipo texto
    - Es recomendable meterlo siempre entre comilla dobles, aunque si el atributo no contiene espacios, se pueden omitir
- Si hay varios atributos, se separan por espacios

```
<img src="urjc.gif" alt="Logo de la URJC">
```

- El nombre del atributo no se puede repetir dentro del mismo elemento. Sí puede aparece el mismo nombre de atributo en un elemento distinto
- Los atributos no están ordenados, no hay garantía de que se mantenga el orden

Elemento head

El elemento head es la cabecera del documento.
Es un contenedor para metadatos del documento HTML. Esta información nunca se muestra directamente. Sus elementos son

`<title>, <base>, <style>, <link>, <meta>, <script>`

- El elemento `<title>` define el título del documento. Es de inclusión obligatoria
- El elemento `<base>` define la base de las direcciones relativas
- Los elemento `<style>` y `<link>` especifican las hojas de estilo CSS
- El elemento `<meta>` se usa para añadir diversa metainformación
- El elemento `<script>` contiene código JavaScript, o un enlace a una página con el código

CSS

CSS (*Cascading Style Sheets*) es un lenguaje de diseño gráfico para crear hojas de estilo, que son son una sucesión de reglas que especifican el formato gráfico de un documento
Las hojas CSS pueden ubicarse

- En el propio documento HTML, dentro del elemento `<style>`

```
<style>
  p {
    background-color: salmon;
  }
</style>
```

En versiones anteriores de HTML se escribía `<style type="text/css">`, pero en HTML5 el atributo *type* en el elemento *style* es obsoleto.

- En un documento distinto, especificando con el elemento `<link>`

  Ejemplo: `<link href="css/bootstrap.min.css" rel="stylesheet">`
  - Es de tipo *void*
  - Puede aparecer varias veces, pero solo en la sección *head* nunca en *body*
  - No confundir con los enlaces a otros documentos HTML dentro del cuerpo del documento, que se indican con `<a>`

Elemento meta

Contiene diversos atributos con metainformación

- El único obligatorio es charset

```
<head>
  <meta charset="UTF-8">
  <meta name="description" content="Tutorial sobre tecnologías web">
  <meta name="keywords" content="HTML,CSS,Bootstrap,JavaScript">
  <meta name="author" content="Juan García">
</head>
```

Codificación de caracteres

En HTML antiguo lo habitual era emplear la codificación ISO-8859. En europa occidental, ISO-8859-1, también llamada latin1. O más bien windows-1252, que es muy similar

- En la actualidad la recomendación es usar UTF-8
- UTF-8 es una forma de codificar unicode, la más habitual pero no la única

La sintaxis de HTML 4 era muy farragosa

```
<head>
<META http-equiv="Content-Type" content="text/html;
↪  charset=ISO-8859-1">
...
</head>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
...
</head>
```

En HTML 5 :

```
<meta charset="UTF-8">
```

Problema: Hay varios lugares donde indicar la codificación

- Desde HTTP (tal y como lo haya configurado el administrador del servidor web)
- Dentro del HTML (tal y como lo configure el autor de la página)

Ambas informaciones pueden ser discrepantes. El convenio es dar precedencia a HTTP. Problema: un servidor que tenga mezcladas páginas HTML 4 (normalmente en ISO-8859-1) y HTML 5 (normalmente UTF-8)

- Se puede configurar el servidor para usar una codificación distinta para cada fichero o directorio, pero el autor y el administrador suelen ser personas distintas, no siempre bien coordinadas

Cuerpo del documento

El elemento body contiene el cuerpo del documento, su contenido principal

- El contenido del cuerpo incluye parrafos de texto, hiperenlaces (también llamados hipervínculos y enlaces), imagénes, tablas, listas, etc
- Solo puede haber un elemento body
- En HTML 4 tenía atributos como *background*, *bgcolor*, *link* o *text* para modificar los colores, pero no se admiten en HTML 5

Elementos de bloque / en línea

En el cuerpo del documento, los elementos pueden ser de dos tipos

- *Block elements*
  Siempre empiezan por nueva línea, y se muestran con cierto margen antes y después del elemento
  Ejemplos: *h1*, *h2*, ... *h6*, *table*, *form*, *ul*, *ol*
- *Inline elements*
  No empiezan con una nueva linea, ocupan el espacio que necesitan, pero sin margen adicional
  Ejemplos: *span*, *a*, *em*, *label*, *img*

p

Párrafo: secuencia de oraciones con unidad temática. Acaba en punto y aparte
El elemento `<p>` crea un párrafo
Entre un párrafo y otro hay

- Un salto de línea
- Una separación adicional

La etiqueta de cierre `</p>` es opcional, si no aparece, se considera implícita antes de la apertura del siguiente elemento. Pero es preferible cerrar explícitamente con la etiqueta de cierre

- Dentro de un párrafo no puede haber otro elemento de nivel de bloque (p.e. *h1*, *table*, *form*, *ul*) si intentamos abrir otro elemento de nivel de bloque, se considera que estamos cerrando implícitamente el párrafo
- Dentro de un párrafo solo puede haber elementos *inline* (p.e. *span*, *a*, *em*, *label*, *img*)

br

La etiqueta `<br>` define el elemento *breaking line*

- En un documento HTML, los saltos de línea se ignoran
- Si queremos forzar un salto de línea (sin definir un párrafo), usamos `<br>`
- En cuanto al formato, es similar a un nuevo párrafo, pero solo con el salto de línea, sin la separación adicional
- Es de tipo *void* (no puede tener ni contenido ni etiqueta de cierre)

em

La etiqueta `<em>` define el elemento *emphasized*

- Típicamente el navegador mostrará el texto en cursiva

No confundir con la unidad *em*, que en español traduciríamos como *eme*, esto es, la letra eme

- Es una unidad clásica de tipografía, para referirse al ancho de la M mayúscula. `1em`, `1.2em`, etc
- En la actualidad, las M mayúsculas suelen ser un poco más estrechas que un em. Así que *em* en la práctica significa *letra de tamaño normal*

pre

La etiqueta `<pre>` define el elemento *texto preformateado*

- El navegador muestra el texto respetando los espacios entre palabras y los saltos de línea
- Normalmente se usa una fuente de ancho fijo, típicamente courier

h1-h6

Las etiquetas `<h1>`, `<h2>`, ... `<h6>`, definen elementos *heading* (encabezado)

- Están organizados jerárquicamente, siendo h1 el más importante y h6 el menos

a

La etiqueta `<a>` define el elemento *anchor* (ancla), que sirve para hacer anclas y también para hacer hiperenlaces
Su nombre es poco afortunado, no describe lo que hace

- La invención del hipertexto se atribuye a D.Engelbart y T.Nelson, por separado, en 1962/1963
- En el hipertexto original solo había *anchors*: enlaces al documentos dentro de la misma sede
- Tim Berners-Lee inventa los enlaces a documentos en otros sitios (la WWW). Pero en el lenguaje HTML mantiene el nombre *anchor*, y por tanto, usa la etiqueta `<a>`
- En la actualidad, un *anchor* es un enlace desde una parte de un documento a otra parte del mismo documento. Pero el elemento `<a>` se usa para hiperenlaces

- Normalmente el elemento `<a>` se usar para hacer hiperenlaces, con el atributo *href (hypertext reference)*

```
<a href="https://www.urjc.es">Página de la URJC</a>
```

- También se pueden enlazar un correo electrónico

```
<a href="mailto:jperez@miempresa.com?Subject=Contacto%20web">
    Envíame un correo</a>
```

(Aunque hoy esto no es recomendable porque la dirección de correo queda expuesta a los spammers)

- O enlazar un script

```
<a href="javascript:alert('¡Hola, mundo!');">Holamundo en
↪  JavaScript</a>
```

Los enlaces pueden ser

- Absolutos a una dirección

```
<a href="http://linkedsite/url.html">Documento</a>
```

- Absolutos dentro del mismo sitio

```
<a href="/url.html">Documento en mismo sitio web</a>
```

- Relativos

```
<a href="url.html">Documento en mismo sitio web y "dir"</a>
```

anchors

Un anchor es una referencia a un punto concreto dentro de un documento

- Un anchor tiene un nombre, con las mismas reglas de cualquier otro atributo
- Para usar un anchor, a su nombre se le antepone la almohadilla
- Ejemplo de dirección con anchor:

```
https://gsyc.urjc.es/~mortuno/index_at.html#evaluacion
```

- Ejemplo de hiperenlace con anchor absoluto

```
<a href="https://www.urjc.es/universidad/org#rector>rector</a>
```

- Ejemplo de hiperenlace con anchor relativo

```
<a href="#inicio>inicio</a>
```

Anchor al estilo HTML 4

En HTML 4, un anchor se creaba definiendo un elemento `<a>`

- Sin atributo href
- Con atributo name

Ejemplo

```
<a name="punto07">Esto es el punto 7</a>

[.....]

<a href="#punto07">Volver al punto 7</a>
```

En HTML 5, esto sigue funcionando en los navegadores, pero es obsoleto, no deberíamos usarlo

Anchor al estilo HTML 5

En HTML 5, un anchor se crea añadiendo el atributo *id* a cualquier elemento contenedor *p, h1, div, pre...*

```
<h1 id="punto07">Punto 7</h1>

[.....]

<a href="#punto07">Volver al punto 7</a>
```

- Naturalmente, estos anchors pueden emplearse tanto de forma relativa como de forma absoluta
- Esta forma de crear anchors también solía funcionar en los navegadores HTML 4, aunque no era la más común

Atributo target

Añadiendo a un enlance el atributo `target="_blank"`, este se abrirá en una nueva pestaña del navegador

```
<a href="http://urjc.es" target="_blank">Abrir en nueva pestaña</a>
```

- En HTML 4 el atributo target tenía otros posibles valores relacionados con el uso de marcos (*_self*, *_parent*, *_top*, *framename*), pero en HTML 5 desaparecen los marcos

div

La etiqueta `<div>` define una división o sección dentro del documento

- Su uso habitual es el de contenedor genérico: delimita un bloque de texto, al que luego se le dará formato mediante reglas CSS

Ejemplo

```
<div class="respuesta">Todas son falsas</div>
```

En HTML 5, además de este elemento se definen otros con el mismo propósito, pero con una semántica más específica

- `<section>`, `<nav>`, `<article>`, `<aside>`, `<hgroup>`, `<header>`, `<footer>`, `<time>`, `<mark>`

http://diveintohtml5.info/semantics.html#new-elements

span

La etiqueta `<span>` (espacio, longitud, lapso) define una división o sección dentro del documento

- Muy similar a div, pero no crea un bloque nuevo y por tanto, no crea una nueva línea
  - Y como no crea línea, no se pueden aplicar atributos de alineación
    `text-align: left | right | center | ... etc`
- Se usa típicamente para dar formato a un grupo de palabras

table, th, tr, td

Las etiquetas `<table>` (tabla), `<th>` (table header), `<tr>` (table row), `<td>` (table data) permiten crear tablas

```
<table>
  <tr>
    <th>Cabecera, primera columna</th>
    <th>Cabecera, segunda columna</th>
  </tr>
  <tr>
    <td>Primera fila, primera columna</td>
    <td>Primera fila, segunda columna</td>
  </tr>
  <tr>
    <td>Segunda fila, primera columna</td>
    <td>Segunda fila, segunda columna</td>
  </tr>
</table>
```

ol,li

Las etiquetas `<ol>` (ordered list), `<li>` (list item), permiten crear listas numeradas

```
<ol>
  <li>Sota</li>
  <li>Caballo</li>
  <li>Rey</li>
</ol>
```

- Las listas numeradas usan, por omisión, números naturales para cada item
- Añadiendo el atributo *type* al elemento *ol* se puede cambiar el tipo de marcador
  Los valores posibles son `1`, `A`, `a`, `I`, `i` para emplear números, letras mayúsculas, minúsculas, números romanos en mayúsculas y en minúsculas, respectivamente

ul,li

Las etiquetas `<ul>` (unordered list), `<li>` (list item), permiten crear listas sin numerar

```
<ul>
  <li>Sota</li>
  <li>Caballo</li>
  <li>Rey</li>
</ul>
```

dl,dt,dd

Las etiquetas `<dl>` (description list), `<dt>` (description term), `<dd>` (description), permiten crear listas de descripciones o definiciones de términos

```
<dl>
   <dt>
      Nombre
   </dt>
   <dd>
      Juan García
   </dd>
   <dt>
      Centro de origen
   </dt>
   <dd>
      ESTIT-URJC
   </dd>
</dl>
```

Ejemplos de tablas y listas:
`http://ortuno.es/tablas_listas.html`

img

La etiqueta `<img>` permite insertar imágenes
Ejemplos

```
<img src="urjc.png" alt="Logo de la URJC">
<img src="urjc.png" alt="Logo de la URJC" width="300" height="240">
```

Este elemento tiene dos atributos obligatorios

- *src*, que especifica el origen del fichero
- *alt*, que indica una descripción en texto del contenido de la imagen, para los navegadores sin gráficos

- El atributo *width* permite indicar el ancho en pixeles de la imagen en pantalla. Puede ser conveniente que la imagen original tenga un ancho algo mayor (para tener flexibilidad en el diseño). Pero evita que sea muy superior
- Es muy habitual incluir una imagen dentro de un elemento `<a>`, de esa forma la imagen se convierte en un hipervínculo

```
<a href="https://www.urjc.es">
   <img src="images/urjc.png" width="120" alt="logo de la URJC">
</a>
```

Observa que normalmente indicaremos el *path* (trayecto) del fichero con la imagen de manera relativa. El trayecto no empieza por el carácter *barra* (/)

- `urjc.png`

  En el mismo directorio donde está este html, hay un fichero llamando *urjc.png*

- `images/urjc.png`

  En el mismo directorio donde está este html, hay un directorio llamado *images*, y dentro, un fichero llamando *urjc.png*

- `../practica03/urjc.png`

  El directorio padre del directorio donde está este html, hay un directorio llamado *practica03*, y dentro, un fichero llamado *urjc.png*

Si anteponemos la barra, el significado cambia por completo

- `/urjc.png`

  En el directorio raiz, el fichero *urjc.png*

- `/images/urjc.png`

  En el directorio raiz, el subdirectorio llamado *images*, y dentro, un fichero llamando *urjc.png*

- `/practica03/urjc.png`

  El directorio raiz, el directorio *practica03*, y dentro, un fichero llamado *urjc.png*

El directorio raiz será:

- Si el fichero lo leemos localmente, el directorio raiz de nuestro sistema de ficheros (disco duro)
- Si el fichero se exporta via web, el directorio raiz establecido por el servidor web,

Naturalmente, estas mismas ideas sobre los trayectos relativos y absolutos, son aplicables en cualquier lenguaje de programación y a cualquier fichero: una imagen jpg, una librería, etc

- Observa que un *path* que incluya la dirección absoluta del usuario casi siempre es un error muy severo, porque deja de funcionar en cuanto cambia el usuario o la máquina

```
/home/alumnos/jperez/images/urjc.png
```

form

Un formulario HTML es un elemento que permite aceptar entrada de información por parte del usuario.

```
<form>
    (Elementos del formulario)
</form>
```

Los elementos posibles son varios

- El principal es el elemento `<input>` que puede de ser de diferentes tipos (`text, password, radio, checkbox`) entre otros
- Otros elementos son

  `<fieldset>` , `<select>` , `<textarea>` y `<button>`
- HTML5 añade los elementos `<datalist>` y `<output>`

Ejemplos de formularios:
`http://ortuno.es/form`

input: text, password, submit

`input` es un elemento HTML de tipo void

- El atributo *name* indica el nombre de campo
- Con el atributo *value* se pueden asignar valores por omisión
- El input de tipo *submit* es el botón *enviar*. Se puede cambiar su texto con el atributo *value*

```
<form action="/action_page.html">
  Nombre de usuario:<br>
  <input type="text" name="usuario" ><br>
  Contraseña:<br>
  <input type="password" name="contrasenya" ><br><br>
  País:<br>
  <input type="text" name="pais" value="España" ><br><br>

  <input type="submit">
</form>
```

Observa que

- Para el *name* no podemos usar letras españolas como la eñe (porque este será el nombre de una variable en el código del servidor que procese el formulario)
- Para el *value* sí podemos. Este será el valor de una variable en el código

input: radio

`<input type="radio">` define un *radio button*, que permite elegir una (y solo una) opción entre varias

```
<form>
  <input type="radio" name="os" value="Linux" checked>Linux<br>
  <input type="radio" name="os" value="macOS" >macOS<br>
  <input type="radio" name="os" value="Windows">Windows<br>
  <input type="radio" name="os" value="other">Otro<br>
</form>
```

input: checkbox

Checkbox es un tipo de input que permite elegir 0 o más opciones de una lista

```
<form>
  <input type="checkbox" name="terminos" value="si">
        He leido los términos y condiciones<br>
  <input type="checkbox" name="publicidad" value="si">
        Deseo recibir comunicaciones comerciales<br>
</form>
```

input: tipos de HTML5

HTML 5 añade nuevos tipos de input

- color (no soportado por todos los navegadores)
- date (no soportado por todos los navegadores)
- datetime-local (no soportado por todos los navegadores)
- email
- month
- number
- range
- search
- tel
- time
- url
- week

fieldset

Un conjunto de entradas se pueden agrupar en un `<fieldset>`, con un título indicado en un elemento `<legend>`

```
<form>
  <fieldset>
    <legend>
      Datos personales
    </legend>

    Elija un color:
    <input type="color" name="favcolor">
    <br> Fecha de nacimiento:
    <input type="date" name="nacimiento">
    <br> Fecha y hora de nacimiento:
    <input type="datetime-local" name="nacimiento-hora">
    <br> E-mail:
    <input type="email" name="email">
    <br> Indica un número del 1 al 10:
    <input type="number" name="numero" min="1" max="10">
    <br>
    <input type="submit">
  </fieldset>
</form>
```

select

El elemento `<select>` permite elegir una opción entre varias

```
<form>
   Indique el departamento:
   <select name="departament">
      <option value="sales">Comercial</option>
      <option value="technical">Técnico</option>
      <option value="webmaster">Webmaster</option>
   </select>
   <input type="submit">
</form>
```

El elemento `<datalist>` es similar pero permite que el usuario escriba una respuesta distinta a las propuestas

textarea

Con el elemento `<textarea>` el usuario puede introducir varias líneas de texto

```
<form>
   <textarea name="mensaje" rows="10" cols="30">
   Escriba aquí su mensaje.
   </textarea>
</form>
```

label

Para que el usuario sepa qué es cada elemento de un formulario, se puede usar:

- Texto HTML ordinario
- Un elemento `<label>`
  Ventajas:
  - Haciendo clic sobre esta etiqueta, se activa el elemento
  - Facilita su interpretación en entornos distintos a navegadores tradicionales, p.e. lectores de HTML
  - Facilita el estilo consistente

Para usar `<label>` hay que

- Añadir un atributo `<id>` al elemento
- Poner en el label un atributo `for` cuyo valor sea el del id

```
<form>
  <label for="ciudad">Ciudad de procedencia:</label>
  <input type="text" name="ciudad" id="ciudad">
  <input type="submit">
</form>
```

- `name` lo use el servidor
- `id` lo usa el navegador

Típicamente se hace que coincidan pero son independientes, no tienen por qué ser iguales

En el caso del `<input type="radio">` y el `<input type="checkbox">`

- Normalmente es innecesario usar `<label>`, el texto dentro del elemento suele ser bastante descriptivo
- Si queremos incluir un `<label>`, se usa como en cualquier otro elemento, un `<label>` por cada input
  - Aunque es recomendable que en el HTML escribamos el `<label>` después del `<input>`, para que el navegador muestre la etiqueta a la derecha del `<input>`, no a la izquierda

Elementos y Atributos obsoletos en HTML5

En HTML 4 había muchos elementos y atributos relacionados con el formato. Algunos de los más habituales:

- align (left, right, center)
- color
- font
- u (underline)

Todos ellos han desaparecido en HTML 5, en su lugar debe usarse CSS [1]

[1] En el elemento span sigue siendo legal usar atributos gráficos, pero no es recomendable. Siempre es preferible CSS

Entities

Las *entities* se usan para

- Representar caracteres que coinciden con metacaracteres de HTML. Es la única forma de indicar caracteres como <
- Representar caracteres que el desarrollador no tenga en su teclado. Su uso es opcional, con UTF-8 siempre se puede escribir cualquier caracter (mediante teclados virtuales o copiando y pegando desde otro sitio)

Cada entity tiene un nombre y un número, se puede usar cualquier de las dos formas

- *Ampersand*, nombre, punto y coma

  ```
  &lt;
  ```

- *Ampersand*, almohadilla, número, punto y coma

  ```
  &#60;
  ```

Algunas entities habituales

- <

```
&lt;      &#60;
```

- >

```
&gt;      &#62;
```

- €

```
&euro;    &#8364;
```

- Ñ

```
&Ntilde;  &#209;
```

- ñ

```
&ntilde;  &#241;
```

Otra entity frecuente es ` ` *non-breaking space*
Es un espacio que:

- Siempre se representa
- Nunca se usa para partir una linea. Ejemplo: para escribir 2 € con garantías de que ambos símbolos estarán en la mísma línea:

```
2 &euro;
```

Uso del editor

- Cualquier documento HTML de tamaño mediano tiene muchos elementos dentro de otros
- Sin la ayuda del editor, sería difícil anidar las etiquetas correctamente

Aqui recomendamos *atom*, un editor muy potente, software libre, disponible para Linux, Windows y macOS

- Cuando llevamos el cursor a una etiqueta, atom subraya en azul esa etiqueta y la etiqueta de cierre/apertura que le corresponda
- Ctrl k Ctrl 1 muestra los elementos de nivel 1
  Ctrl k Ctrl 2 muestra los elementos de nivel 3
  (Pulsar Ctrl y k simultáneamente, luego pulsar Ctrl y el número)
  Etc

El *plug-in* beautify de atom permite tabular jerárquicamente el código
Podemos instalarlo usando apm, *atom package manager*
Desde la shell ejecutamos

```
apm install atom-beautify
```

Uso:

1. Seleccionamos el texto a tabular
2. Ctrl alt b

Material complementario

- HyperText Markup Language (Wikibook):
  `http://en.wikibooks.org/wiki/HTML_Programming`
- HTML5: A tutorial for beginners:
  `http://www.html-5-tutorial.com/`
- Dive into HTML5:
  `http://diveintohtml5.info`
- HTML5 (Wikipedia):
  `http://en.wikipedia.org/wiki/HTML5`
- Web Fundametals (Code Academy):
  `http://www.codecademy.com/tracks/web`

Hojas de estilo CSS

El material didáctico “CSS - Hojas de estilo”
está basadas en el libro de uniwebsidad.com
disponible en `http://www.uniwebsidad.com/libros/css`
Se ha pedido explícitamente autorización al autor original
para realizar esta obra derivada con fines educativos
(c) Javier Eguiluz - uniwebsidad.com

Derivado a partir de material de
Jesús M. González-Barahona y Gregorio Robles.
El original está disponible en
`http://cursosweb.github.io`

¿Qué es CSS?

- CSS, *Cascading Style Sheets* es un lenguaje de hojas de estilos creado para **controlar el aspecto** o presentación de los documentos electrónicos definidos con HTML
- Es la mejor forma de **separar los contenidos y su presentación** y es imprescindible para crear páginas web complejas
  - Obliga a crear documentos HTML bien definidos y con significado completo (también llamados *documentos semánticos*)
  - Mejora la accesibilidad del documento
  - Reduce la complejidad de su mantenimiento
  - Permite adaptar el documento a dispositivos distintos, manteniendo el código HTML

Antes del CSS

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<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>
```

Con CSS

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Ejemplo de estilos con CSS</title>
  <style>
    h1 { color: red;  font-family: Arial;   font-size: large;  }
    p  { color: gray; font-family: Verdana; font-size: medium; }
  </style>
</head>

<body>
  <h1>Titular de la página</h1>
  <p>Un párrafo de texto no muy largo.</p>
</body>
</html>
```

CSS en un documento HTML

Se pueden integrar instrucciones CSS de varias maneras en un documento HTML:

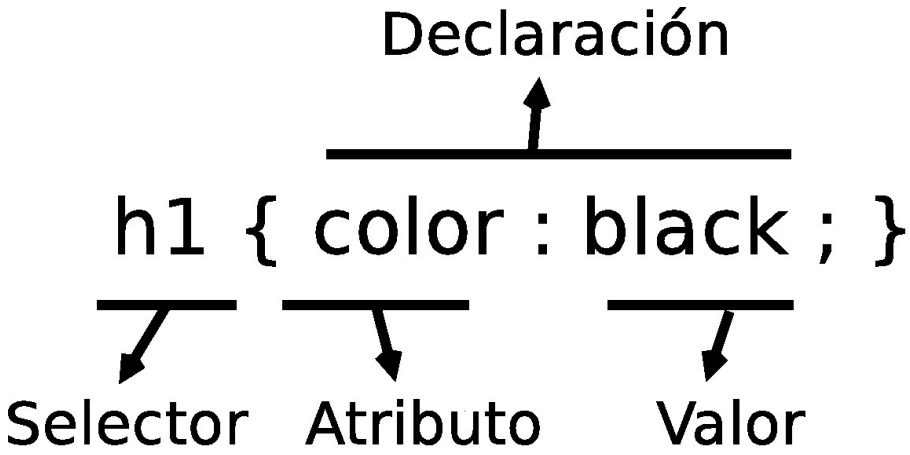
1. Hoja de estilos externa
   Con un elemento *link* en la cabecera, que apunte a un fichero .css que contenga las reglas
   Recomendable para la mayoría de los casos

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

2. Hoja de estilos interna
   Escribir las reglas CSS dentro de un elemento *style*, en la cabecera del documento HTML
   Adecuado para ejemplos sencillos
3. Reglas CSS insertadas directamente en los elementos HTML
   Un atributo *style* directamente en cada elemento, cuyo valor sea la regla
   No recomendable

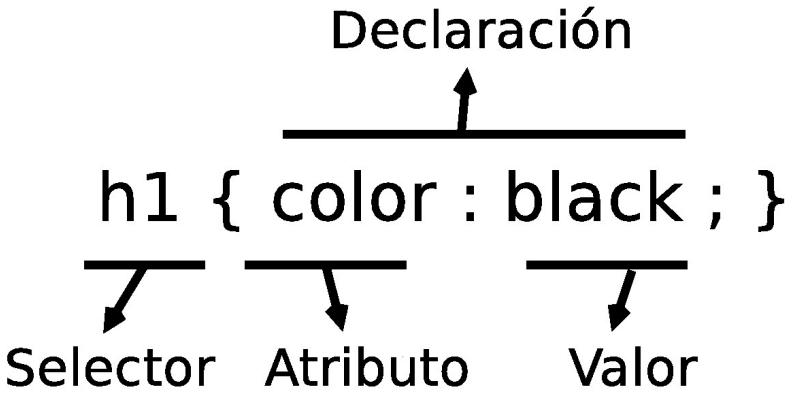
https://www.w3schools.com/css/css_howto.asp

Glosario Básico (I)



- **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.

Glosario Básico (y II)



- **Declaración**: especifica los estilos que se aplican a los elementos. Está compuesta por una o más atributos CSS.
- **Atributo**: 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.

CSS3 define (unos pocos) cientos de atributos

Selectores

- A un mismo elemento HTML se le pueden aplicar varias reglas
- Cada regla puede aplicarse a un número ilimitado de elementos
- Cuando el selector de dos o más reglas CSS es idéntico, se pueden agrupar las declaraciones de las reglas para hacer las hojas de estilos más eficientes
- Muy importante: Cuando se dispara una regla para un elemento, los valores de los atributos se aplican a ese elemento y a todos sus descendientes
  - Naturalmente, si un elemento asigna valor a un atributo heredado, prevalece el valor indicado explícitamente, la herencia queda sin efecto

- CSS 2.1 incluye una docena de tipos diferentes de selectores, que permiten seleccionar de forma muy precisa elementos individuales o conjuntos de elementos dentro de una página web.
- Los selectores de CSS tienen mucha utilidad porque también se usan en JavaScript

Resumen:

	Significado
(espacio)	descendiente (hijo, nieto...)
.	clase
,	OR
(concatenación)	AND
#	id

Selectores básicos

1. Selector universal
2. Selector de tipo o etiqueta
3. Selector descendiente
4. Selector de clase
5. Selector de identidad

Selector Universal

- No se utiliza habitualmente
- Generalmente es equivalente para poner estilo a $< body >$
- Se suele combinar con otros selectores y además, forma parte de algunos hacks muy utilizados

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

Selector de tipo o etiqueta

- Selecciona todos los elementos de la página cuya etiqueta HTML coincide con el valor del selector
- Se pueden agrupar todas las reglas individuales en una sola regla con un selector múltiple. La coma significa *or*
- Buena práctica: agrupar los atributos comunes de varios elementos en una única regla CSS y posteriormente definir los atributos específicos de esos mismos elementos

```
h1, h2, h3 {
  color: #8A8E27;
  font-weight: normal;
  font-family: Arial, Helvetica, sans-serif;
}

h1 { font-size: 2em; }
h2 { font-size: 1.5em; }
h3 { font-size: 1.2em; }
```

Selector descendiente

- Selecciona los elementos contenidos dentro de otros elementos.

Ejemplo: elementos span contenidos dentro de elementos p

```
p span { color: red; }
[...]
<p>
  ...
  <span>Texto1</span>
  <a href="">...<span>Texto2</span></a>
  ...
</p>
```

Texto 1 evidentemente cumple la regla.
Texto 2 es un elemento span contenido dentro de un enlace contenido dentro de un elemento p. Por tanto, Texto 2 están contenido dentro de un p. Aunque no sea descendiente directo, es descendiente y la regla se aplica.

Ejercicio

¿Qué elementos se seleccionarían con estos tipos de selectores?

- p a span em { text-decoration: underline; }
- p, a, span, em { text-decoration: underline; }
- p a { color: red; }
- p * a { color: red; }

Selector de clase

- Se utiliza el atributo class de HTML sobre ese elemento para indicar directamente la regla CSS que se le debe aplicar
- Se crea en el archivo CSS una nueva regla llamada destacado con todos los estilos que se van a aplicar al elemento
- Se prefija el valor del atributo class con un punto (.)

```
.destacado { color: red; }
[...]
<p class="destacado">
  Lorem ipsum dolor sit amet...
</p>
<p>Nunc sed lacus et
  <a href="#" class="destacado">est adipiscing</a>
</p>
<p>Class aptent taciti <em class="destacado">sociosqu ad</em>
</p>
```

Esta regla se aplica a cualquier elemento de clase *destacado*

El atributo clase puede tener varios valores, separados por espacios

```
<style>
 .comentario {color : blue;}
 .noticia {color : red;}
 .obsoleto {text-decoration : line-through;}
</style>
[...]
<body>
  <p class="comentario"> Lorem ipsum dolor sit amet<p>
  <p class="comentario obsoleto"> consectetur adipisicing elit, </p>
  <p class="noticia "> sed do eiusmod tempor incididunt </p>
  <p class="noticia obsoleto"> ut labore et dolore magna aliqua.</p>
</body>
```

Ejemplo:
http://ortuno.es/ej000_clases.html

Selector de clase más específico

- Combinando el selector de tipo y el selector de clase, se obtiene un selector mucho más específico.

```
p.destacado { color: red }
[...]
<p class="destacado">
  Lorem ipsum dolor sit amet...
</p>
<p>Nunc sed lacus et
  <a href="#" class="destacado">est adipiscing</a>
</p>
<p>Class aptent taciti <em class="destacado">sociosqu ad</em>
</p>
```

Esta regla se aplica a los elementos de tipo párrafo, que además sean de clase *destacado*. (En este ejemplo, solo una vez)

```
.a.b  {...}
```

Esta regla se aplica a los elementos de clase *a* que además sean de clase *b*

- Es equivalente a decir los elementos de clase *b* que además sean de clase *a* (las clases son atributos, y los atributos no tienen orden)
- Por supuesto, también se aplica a todos sus descendientes

`http://ortuno.es/concatenacion_clases.html`

Ejercicio

¿Qué elementos se seleccionarían con estos tipos de selectores?

- p.aviso { ... }
- p .aviso { ... }
- p, .aviso { ... }
- *.aviso { ... }

Selectores de identificador

- Aplica estilos CSS a un único elemento de la página
- El identificador ha de ser único: dos elementos distintos no pueden tener el mismo identificador. Y un elemento no puede tener dos identificadores

```
#destacado { color: red; }

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

Ejercicio

¿Qué elementos se seleccionarían con estos tipos de selectores?

- p#aviso { ... }
- p #aviso { ... }
- p, #aviso { ... }
- *#aviso { ... }

Ejercicio: Combinación de selectores

¿Qué elementos se seleccionarían con estos tipos de selectores?

- .aviso .especial { ... }
- div.aviso span.especial { ... }
- ul#menuPrincipal li.destacado a#inicio { ... }

Colisión de estilos (simplificado)

1. Cuanto más específico sea un selector, más importancia tiene su regla asociada.
2. A igual especificidad, se considera la última regla indicada.

Unidades de medida

- Unidades absolutas
  - in, cm, mm, pt, pc
- Unidades relativas
  - em, ex, px
- Porcentajes

En general, se recomienda el uso de unidades relativas siempre que sea posible

Normalmente se utilizan

- Pixel y porcentajes
  Para definir el *layout* (la distribución) del documento. Esto es, la anchura de las columnas y de los elementos de las páginas
- em y porcentajes
  Para definir el tamaño de letra de los textos

Especificación del color

Hay dos formas principales de indicar el color

- Mediante su nombre
  Los 18 principales son:
  red, cyan, blue, darkblue, lightblue, purple, yellow, lime, magenta, white, silver, gray/grey, black, orange, brown, maroon, green, olive
  Pero hay otros 140 nombres

`https://www.w3schools.com/colors/colors_names.asp`

- Mediante su código hexadecimal
  Se representa por con una almohadilla y 6 dígitos hexadecimales (dos dígitos para el rojo, dos dígitos para el verde y dos dígitos para el azul). De esta forma se pueden especificar los 16 777 216 colores del espacio sRGB
  Ejemplos:

```
Código    Nombre Pantone
----------------------
#ff0000   red
#ff2800   ferrari red
#f5e050   minion yellow
```

Cuidado: los nombres de color del estandar industrial Pantone no coinciden con los nombres de color CSS

Hay muchas herramientas online que facilitan la elección de colores.

- Para elegir un color individual, buscaremos un *html color picker*
- Para elegir el conjunto de colores que emplearemos en una página (paleta de colores), buscaremos un *color palette generator*. Como p.e. `https://coolors.co`

Los principales atributos relacionados con el color son

- color
- backgroud-color
- border-color

Generador de paletas coolors.co

`https://coolors.co`

- Podemos iniciar el generador de paletas sin necesidad de crear cuenta
- Cada vez que pulsemos espacio nos mostrará una nueva paleta de barras verticales, con un componente aleatorio pero siguiendo ciertas normas de diseño gráfico
- Con las flechas del teclado (izquierda/derecha) podemos volver a las paletas generadas previamente
- En la barra donde coloquemos el cursor aparecerán una serie de iconos, llevando el cursor a cada uno de ellos podemos ver su nombre

- Por omisión las paletas son de cinco colores
  - Para reducir el número pulsamos en alguna de las barras el icono *remove color*, con forma de aspa
  - Para aumentar el número, llevamos el cursor a una frontera entre barra y barra y pulsamos el icono *más*
- Cuando nos guste un color, lo podemos fijar con el icono del candado: cuando sigamos pulsando espacio, esa barra se mantendrá
- Pulsando el icono *check contrast* sabremos si el texto negro o blanco sería adecuado para ese fondo
- El icono *view shades* de cada barra, con forma de rejilla, nos permite modificar ese color

Atributos relacionados con el texto

- Alineación

```
text-align: left | right | center | justify | initial | inherit;
```

- Subrayado

```
text-decoration: none | underline | overline | line-through |
                 initial | inherit;
```

- Tamaño

```
font-size: medium | xx-small | x-small | small | large |
           x-large | xx-large | smaller | larger | (length) |
           initial | inherit;
```

- Estilo

```
font-style: normal | italic | oblique | initial | inherit;
```

Atributos relacionados con los bordes

Los atributos del borde de una caja se especifican con:

- border-width
- border-style
- border-color

Puede abreviarse como simplemente *border*

- Ancho del borde

```
border-width: medium | thin | thick | (length) |
              initial | inherit;
```

- Estilo del borde

```
border-style: none | hidden | dotted | dashed | solid | double |
              groove | ridge | inset | outset | initial | inherit;
```

https://www.w3schools.com/css/css_border.asp

Tipos de elementos (I)

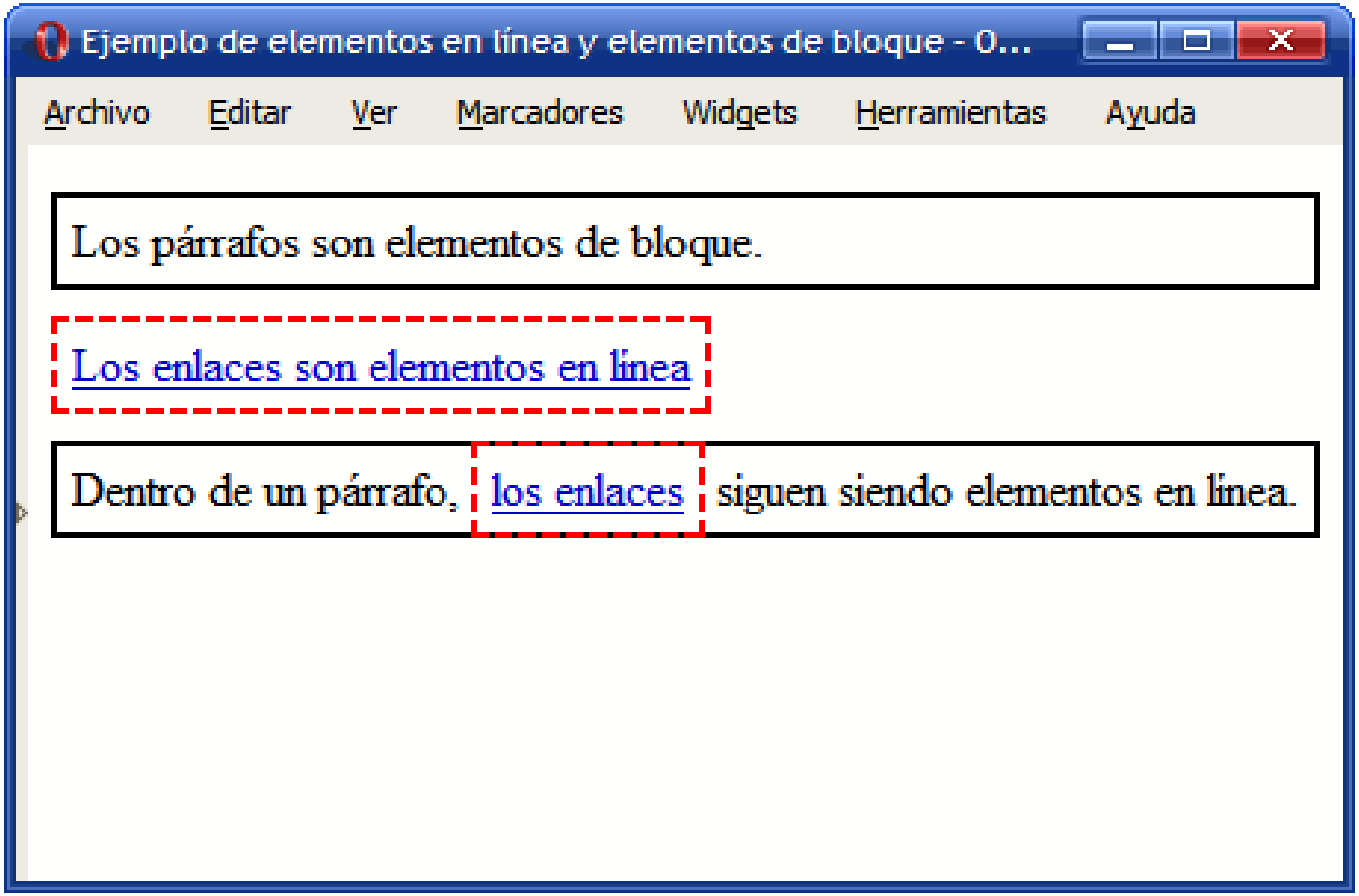
El estándar HTML clasifica a todos sus elementos en dos grandes grupos:
Elementos de línea:

- Los elementos en línea ("inline elements" en inglés) no empiezan necesariamente en nueva línea y sólo ocupan el espacio necesario para mostrar sus contenidos.

Tipos de elementos (II)

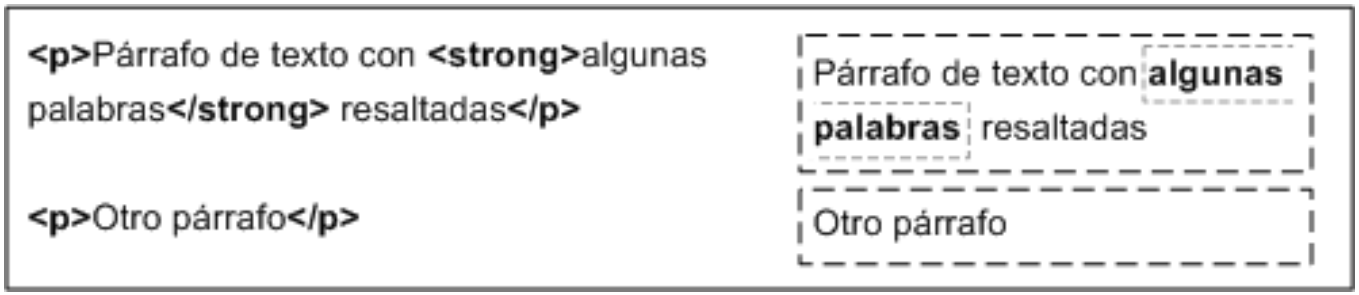
Elementos de bloque:

- Los elementos de bloque ("block elements" en inglés) siempre empiezan en una nueva línea y ocupan todo el espacio disponible hasta el final de la línea



El modelo de cajas

- Es el comportamiento de CSS que hace que todos los elementos de las páginas se representen mediante cajas rectangulares
- Cada vez que se inserta una etiqueta HTML, se crea una nueva caja rectangular que encierra los contenidos de ese elemento



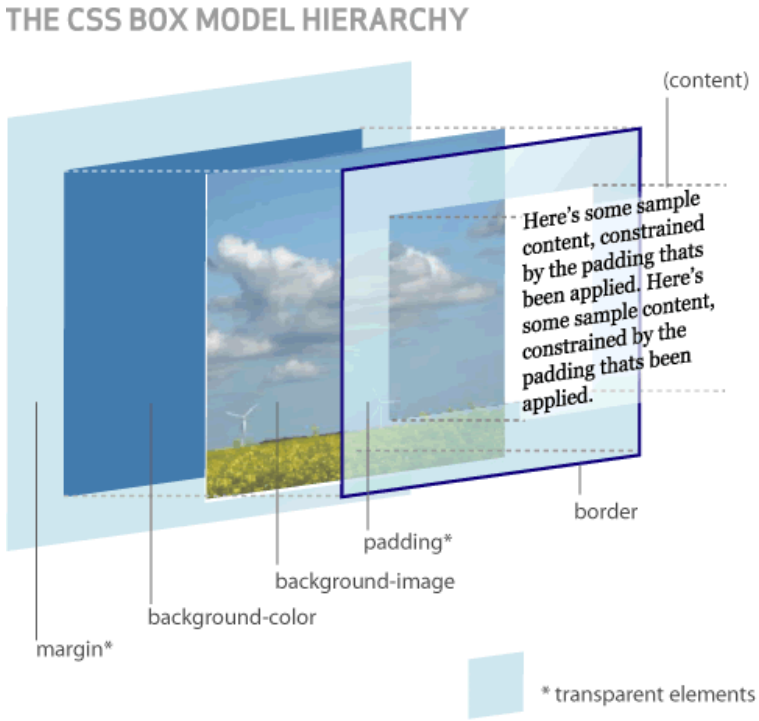
El modelo de cajas (II)

- No son visibles a simple vista porque inicialmente no muestran ningún color de fondo ni ningún borde

Ejemplo de http://www.alistapart.com/ después de forzar a que todas las cajas muestren su borde

El modelo de cajas (III)

- 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:



(Esquema utilizado con permiso de http://www.hicksdesign.co.uk/boxmodel/)

Partes que componen cada caja

- **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 cajas adyacentes.

Margen, relleno, bordes y modelo de cajas (I)

- El margen, el relleno y los bordes establecidos a un elemento determinan la anchura y altura final del elemento

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

Ejemplos:
`http://ortuno.es/cajas.html`

- Si indicamos 4 valores, se refieren a arriba, derecha, abajo, izquierda

```
margin: 8px 10px 8px 8px
```

- Si indicamos 3 valores, se refieren a arriba, ambos lados, abajo

```
margin: 8px 10px 8px
```

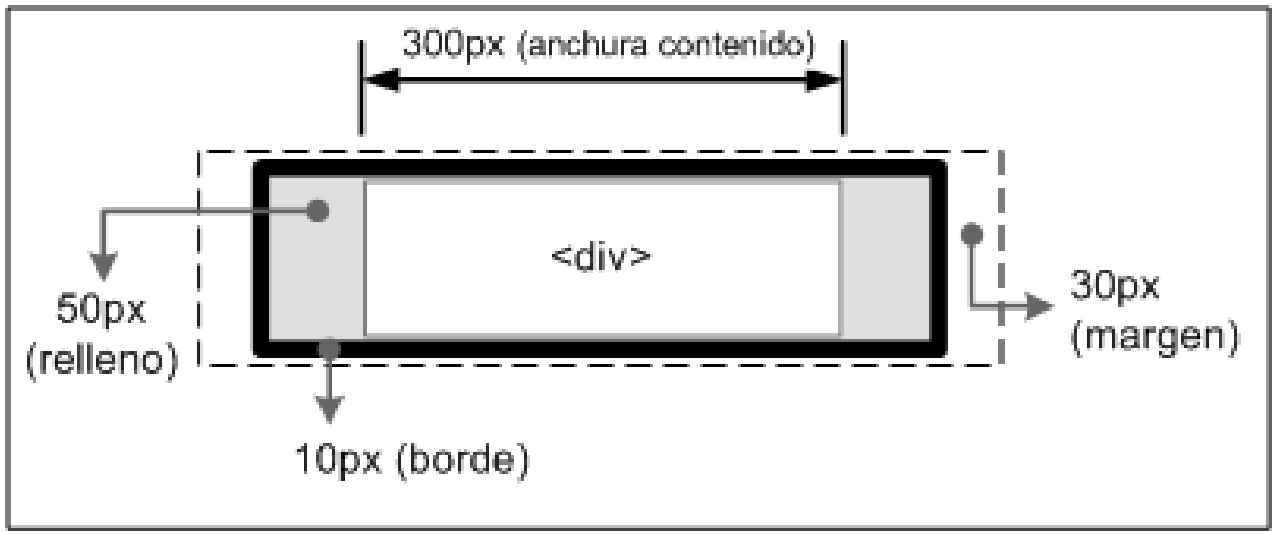
- Si indicamos 2 valores, se refieren a márgenes verticales, márgenes horizontales

```
margin: 8px 10px
```

- Si indicamos 1 valor, se aplica a los 4 márgenes (arriba, derecha, abajo, izquierda, todos iguales)

```
margin: 8px
```

Margen, relleno, bordes y modelo de cajas (y II)

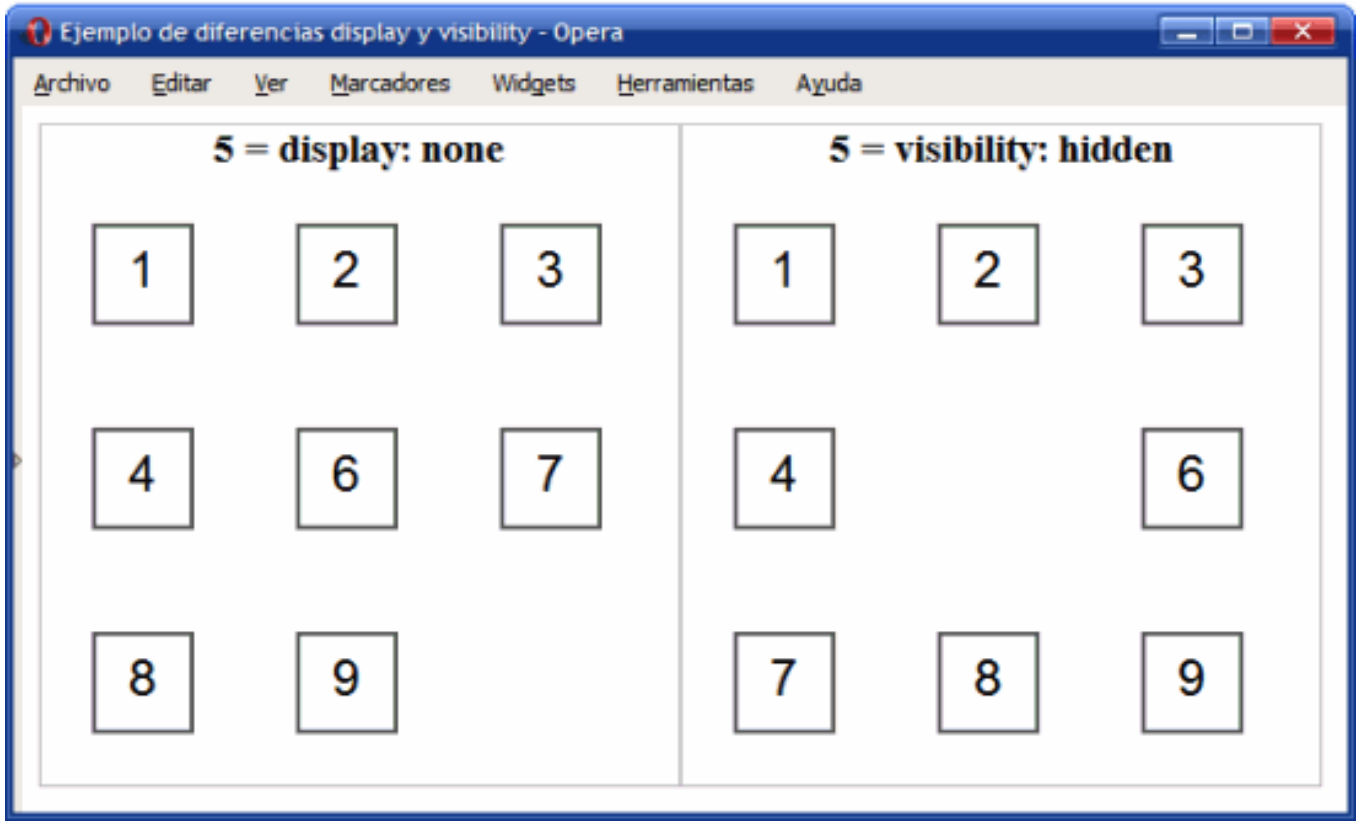


De esta forma, la anchura del elemento en pantalla sería igual a la suma de la anchura original, los márgenes, los bordes y los rellenos: 30px + 10px + 50px + 300px + 50px + 10px + 30px = 480 píxel

Visualización

- CSS define otros cuatro atributos para controlar su visualización: display, visibility, overflow y z-index.
- El atributo display permite ocultar completamente un elemento haciendo que desaparezca de la página. Como el elemento oculto no se muestra, el resto de elementos de la página se mueven para ocupar su lugar.
- El atributo display también permite modificar el comportamiento de un elemento a bloque (block) o en línea (inline).
- El atributo visibility permite hacer invisible un elemento, lo que significa que el navegador crea la caja del elemento pero no la muestra. En este caso, el resto de elementos de la página no modifican su posición, ya que aunque la caja no se ve, sigue ocupando sitio.

Diferencias entre display y visibility



Otros atributos

CSS tiene muchos otros atributos que no veremos aquí

- Normalmente los programadores no sabemos hacer buenos diseños gráficos, no es recomendable empeñarse en configurar el CSS a bajo nivel
- Si realmente quieres usar otros atributos, recuerda que *Google es tu amigo*

Ejemplo: centrar una imagen
`http://ortuno.es/imagen_centrada.html`

Bootstrap 5

¿Qué es Bootstrap?

- Bootstrap es un framework libre para desarrollo web
- Desarrollado inicialmente en 2011 por ingenieros de Twitter
- La versión actual, Bootstrap 5, aparece en mayo de 2021. A diferencia de las anteriores, emplea *vanilla JavaScript*, no *jQuery*
- Incluye plantillas HTML y CSS con tipografías, formas, botones, cuadros, tablas, barras de navegación, carruseles de imágenes y muchas otras
- Aunque su preferencia es *mobile first*, permite crear diseños que se ven bien en múltiples dispositivos (*responsive design*)
- Orientado a programadores, no a diseñadores gráficos
- Es posiblemente la herramienta más popular para este fin, aunque hay alternativas como Foundation

Características de Bootstrap

Ventajas

- Resulta sencillo y rápido escribir páginas con muy buen aspecto
- Se adapta a distintos dispositivos (*responsive design*)
- Proporciona un diseño consistente
- Es compatible con los navegadores modernos
- Es software libre

Inconvenientes

- Al ser una herramienta muy popular, las páginas web que no estén personalizadas *quedan iguales que las de todo el mundo*
- No es especialmente fácil personalizar los estilos (Foundation puede ser más adecuado para esto)

Holamundo en Bootstrap

Para usar Bootstrap basta con

- Definir el *viewport*
- Incluir un elemento *link* apuntando al CSS de Bootstrap
- Incluir un elemento *script* apuntando al código JavaScript de Bootstrap

```html
<!doctype html>
<html lang="es-ES">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">

  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
   integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
   rel="stylesheet" crossorigin="anonymous">
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
   integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p"
   crossorigin="anonymous">
  </script>

  <title>Holamundo en Bootstrap 5</title>
</head>

<body>
  <div class="container">
    <h1>Holamundo en Bootstrap 5</h1>
  </div>
</body>

</html>
```

http://ortuno.es/hola_bootstrap5.html

Adaptación del contenido a la pantalla

Ya desde su diseño original, un requisito importante para el web es que las páginas se pueda representar en pantallas de cualquier tamaño. Con la aparición de los *smartphones*, esto es aún más necesario y más complicado. A lo largo de los años se han usado varias técnicas para conseguir esto, cada vez mejores

1. Técnica inicial
   Viewport. Barras de desplazamiento horizontal y vertical, recomposición de los elementos sobre el *viewport*
2. Primeros *smartphones*
   Viewport virtual
3. Teléfonos móviles actuales
   Diseño *responsive* basado en *grid*

Viewport

Para diseñar webs en dispositivos móviles, es importante tener claro qué es el *viewport* y cómo se comporta

- *Viewport* es la zona visible de una página web. En los navegadores tradicionales de escritorio, coincide con la ventana del navegador
- Supongamos una página web grande y compleja, como la portada de un periódico. La página no cabrá en la ventana del navegador, el usuario usará las barras de scroll para mover el *viewport* sobre el documento. Al redimensionar la ventana, cambiará el tamaño del *viewport*
- Cambiar el tamaño del viewport reposiciona el texto y todos los elementos: las líneas se truncan, las imáges se recolocan, etc

Viewport es un rectángulo donde se compone un fragmento (tal vez completo) de la página web para presentarla al usuario

Viewport virtual

Con la aparición de los navegadores en teléfonos móviles, los cambios del tamaño de la pantalla son mucho más drásticos. La técnicas tradicionales siguen funcionando, pero proporcionan una experiencia de uso muy poco satisfactoria

- El área visible de un móvil es demasiado pequeña, componer una página web tradicional en ese *viewport* queda mal. Observa lo que sucede en esta página antigua cuando la ventana es muy grande o muy pequeña
  `https://tinyurl.com/y7e77lvw`
- Además, en un navegador para móvil no hay barras de scroll, ocuparían un espacio demasiado valioso. Ni ventanas, serían demasiado pequeñas

Para solucionar este problema, aparece el concepto del *viewport virtual*, mayor que el *viewport físico*, esto es, mayor que lo que se puede representar en la pantalla real del dispositivo

- Lo introduce Apple para Safari en iOS, luego pasa a ser estándar
- El ancho del *viewport* virtual es razonablemente grande, por ejemplo 980 pixeles en el navegador safari para iPhone
- El navegador compone la página sobre este viewport virtual, ya no hacen falta barras desplazamiento horizontal
- El usuario arrastra el *viewport físico* (la pantalla real, más pequeña) sobre el viewport virtual, para que le muestre una zona u otra del documento. También se le puede permitir hacer zoom
    - Redimensionar este viewport físico ya no provoca la recomposición de la página

Páginas responsive

Una página web moderna con un minimo de calidad se entiende que tiene que ser *responsive*

- La página se adapta al tamaño de la pantalla (escritorio, tablet, móvil), sin usar la barra de desplazamiento horizontal, que es muy incómoda. La barra de desplazamiento vertical se sigue usando, no es molesta
- El diseño *responsive* tal y como lo conocemos en la actualidad se basa en el uso de un *grid*. En español se traduce por *cuadrícula*, *rejilla* o *casilla* [2]. Aquí veremos el *grid* de Bootstrap, hay otros pero siempre son similares
- En estas páginas ya no hace falta un *viewport* virtual, porque la página está diseñada para adaptarse al *viewport* ordinario (la pantalla pequeña)

[2] en ocasiones le llamaremos celda por analogía con las hojas de cálculo

Las mismas 12 casillas se presenta de forma distinta en un ordenador

XXXXXXXXXXXX

En un tablet

XXXXXX
XXXXXX

En un móvil

XXXX
XXXX
XXXX

Mobile first

- Con una propiedad de etiqueta meta, podemos indicar la escala inicial del *viewport*
- Como las páginas con bootstrap son *responsive*, especificamos que el *viewport* virtual coincida con el ancho de la pantalla, esto es, con el *viewport* ordinario. En otras palabras: que no haya un *viewport* virtual

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

- También se puede inhabilitar el zoom en dispositivos móviles con `user-scalable=no`
- Los usuarios sólo podrán hacer *scroll* y tendrá una apariencia nativa.

```
<meta name="viewport" content="width=device-width, initial-scale=1,
maximum-scale=1, user-scalable=no">
```

Contenedores

- Para que sean *responsivos*,todos los elementos de Bootstrap deben estar dentro de un elemento contenedor
- Los contenedores no se pueden anidar
- Asegúrate de cerrar correctamente cada contenedor. Si alguna fila queda fuera, sus columnas quedarán mal alineadas. Y este error no lo detecta el W3C validator.

Para un contenedor responsivo de tamaño fijo, se usa `.container`

```
<div class="container">
  ...
</div>
```

http://ortuno.es/container.html

Si se desea un contenedor con el ancho total (del *viewport*), se ha de usar `.container-fluid`

```
<div class="container-fluid">
  ...
</div>
```

http://ortuno.es/container_fluid.html

El sistema de cuadrículas de Boostrap

- La pantalla se divide en filas y columnas
- Llamaremos *celda* a la intersección entre una fila y una columna [3]
- El contenido se coloca dentro de una celda, y siempre se mostrará dentro de esa misma celda
- El ancho de cada celda se mide en casillas
- En cada fila hay hasta 12 casillas, que el diseñador decide cómo repartir entre celdas
- Cuando la pantalla tiene la suficiente resolución, las celdas de la misma fila se ven unas al lado de otras (*disposición normal*)
- Cuando la resolución disminuye, las celdas que originalmente estaban en la misma fila, pasan a verse unas encima de otras (*disposición apilada*)

[3] en realidad Bootstrap no usa este concepto, habla solo de columna, pero entendemos que es una terminología confusa: un ladrillo individual no puede ser una columna, hacen falta varios ladrillos apilados

- Cada fila es un elemento *div* de HTML con la clase *row*. Observa que empleando notación de los selectores de CSS (donde el punto significa *clase*), podemos llamarle `.row`
- Dentro de la fila hay elementos a los que en esta asignatura llamamos *celdas*, que pueden ser de los tipos `.col-N`, `.col-sm-N`, `.col-md-N`, `.col-lg-N`, `.col-xl-N` o `.col-xxl-N`

Ejemplo:

```
<div class="row">
  <div class="col-md-4">
  </div>
</div>
```

Estos 6 tipos de celdas dependen del ancho de *viewport* (pantalla) en el que queramos que las celdas se muestren en disposición normal, no apilada

- .col-N
  Pantallas muy pequeñas, de menos de 576px
- .col-sm-N
  Pantallas pequeñas de al menos 576px
- .col-md-N
  Pantallas medianas de al menos 768px
- .col-lg-N
  Pantallas grandes de al menos 992px
- .col-xl-N
  Pantallas muy grandes de al menos 1200px
- .col-xxl-N
  Pantallas extra grandes de al menos 1400px

Donde N es un número entre 1 y 12, que indica el ancho de cada columna. El total de las columnas de cada fila puede sumar un máximo de 12. La frontera entre cada uno de estos tamaños se denomina *breackpoint*

- Columnas *.col-xxl-N*
  Disposición normal en pantallas *extra grandes*.
  Se apilan en pantallas muy grandes, grandes, medianas, pequeñas o muy pequeñas
- Columnas *.col-xl-N*
  Disposición normal en pantallas *muy grandes* o *extra grandes*.
  Se apilan en pantallas grandes, medianas, pequeñas o muy pequeñas
- Columnas *.col-lg-N*
  Disposición normal en pantallas *grandes*, *muy grandes* o *extra grandes*.
  Se apilan en medianas, pequeñas o muy pequeñas

- Columnas *.col-md-N*
  Disposición normal en pantallas *medianas*, *grandes*, *muy grandes* o *extra grandes*.
  Se apilan en: pequeñas o muy pequeñas
- Columnas *.col-sm-N*
  Disposición normal en pantallas *pequeñas*, *medianas*, *grandes*, *muy grandes* o *extra grandes*.
  Se apilan en muy pequeñas
- Columnas *.col-N*
  Disposición normal en cualquier pantalla: *muy pequeñas*, *pequeñas*, *medianas*, *grandes*, *muy grandes* o *extra grandes*.
  Nunca se apilan

Dicho de otro modo

- Cada tipo de columna se muestra en su disposición normal, esto es, horizontalmente, si la pantalla es de su tipo o de un tipo mejor
- En otro caso, las casillas se apilan verticalmente

Esto parece un poco complicado, pero con el siguiente ejemplo verás que no:

1. Vete a `http://ortuno.es/rejilla_01.html`
2. Maximiza la ventana
3. Vete reduciéndo el ancho gradualmente. Esto es equivalente a tener una pantalla menor. Verás que según vayas reduciendo, las cuadrículas que originalmente están en disposición normal (horizontal) se van apilando (verticalmente)

Alineación horizontal de las cuadrículas

Las celdas (que formarán columnas cuando sean varias a la misma distancia del eje vertical) se pueden alinear de diversa forma en horizontal añadiendo a la fila (el *div* de clase *row*) las clases

```
justify-content-start
justify-content-center
justify-content-end
justify-content-around
justify-content-between
justify-content-evenly
```

En el código fuente de este ejemplo podrás ver

- El resultado de usar las distintas clases de alineamiento horizontal. En este caso con dos columnas de 3 casillas cada una
- El uso de la clase *border* con el color *border-primary*

`http://ortuno.es/rejilla_02.html`

Componentes de Bootstrap

Bootstrap viene con una serie de estilos (generalmente en formato de clase CSS) y componentes en JavaScript.

- `btn`
- `table`
- `card`
- `carousel`
- y otras utilidades responsivas

Colores contextuales

La gama concreta de colores se decidirá en el CSS.
Aquí pondremos clases con valor semántico.

- Con alguna excepción como *light* o *white*, puesto que al elegir el color del fondo, puede ser necesario indicar también el color del texto (en este ejemplo, el texto blanco sobre fondo blanco no se ve)

```
<h2>Colores del texto</h2>
<p class="text-muted">Muted (silenciado, apagado).</p>
<p class="text-primary">Primary.</p>
<p class="text-success">Success (éxito).</p>
<p class="text-info">Info.</p>
<p class="text-warning">Warning.</p>
<p class="text-danger">Danger.</p>
<p class="text-secondary">Secondary.</p>
<p class="text-body">Body (típicamente negro).</p>
<p class="text-light">Light grey .</p>
<p class="text-white">White.</p>
```

```
<h2>Colores del fondo</h2>
<p class="bg-primary text-white">Primary.</p>
<p class="bg-success text-white">Sucess (éxito)</p>
<p class="bg-info text-white">Info.</p>
<p class="bg-warning text-white">Warning.</p>
<p class="bg-danger text-white">Danger.</p>
<p class="bg-secondary text-white">Secondary.</p>
<p class="bg-dark text-white">Dark (grey).</p>
<p class="bg-light text-dark">Light (grey).</p>
```

http://ortuno.es/colores.html

Botones

La clase `btn` de Bootstrap puede añadirse a los elementos HTML `<button>`, `<input>` y `<a>`

- Tienen efecto *hover*: destacan un botón cuando se posiciona el ratón encima

```
<button type="button" class="btn">Basic</button>
<button type="button" class="btn btn-primary">primary</button>
<button type="button" class="btn btn-secondary">secondary</button>
<button type="button" class="btn btn-success">success</button>
<button type="button" class="btn btn-info">info</button>
<button type="button" class="btn btn-warning">warning</button>
<button type="button" class="btn btn-danger">danger</button>
<button type="button" class="btn btn-dark">dark</button>
<button type="button" class="btn btn-light">light</button>
<button type="button" class="btn btn-link">link</button>
```

```
<button type="button" class="btn btn-outline-primary">btn-outline-primary</button>
<button type="button" class="btn btn-outline-secondary">btn-outline-secondary</button>
<button type="button" class="btn btn-outline-success">btn-outline-success</button>
<button type="button" class="btn btn-outline-info">btn-outline-info</button>
<button type="button" class="btn btn-outline-warning">btn-outline-warning</button>
<button type="button" class="btn btn-outline-danger">btn-outline-danger</button>
<button type="button" class="btn btn-outline-dark">btn-outline-dark</button>
<button type="button" class="btn btn-outline-light text-dark">btn-outline-light</button>
```

Con el atributo disabled (atributo, no clase), el botón queda inhabilitado

```
<button type="button" class="btn btn-primary" disabled>
  disabled Primary
</button>
```

http://ortuno.es/botones.html

Imágenes

Para modificar el aspecto de una imagen, Bootstrap, nos permite añadir clases al elemento `<img>`
Contorno:

- rounded
  Esquinas redondeadas
- rounded-circle
  Circular
- img-thumbnail
  Miniatura (reborde blanco)

Alineación:

- float-start
  Izquierda
- float-end
  Derecha
- mx-auto d-block
  centrada
- fluid
  Todo el espacio disponible

```
<div class="row">
  rounded
  <div class="col-xl-12">
    <img src="images/plaza_espana.jpg" alt="Plaza de España,
    ↪  Madrid"
     width="400" class="rounded">
  </div>
</div>
```

http://ortuno.es/imagenes.html

Tablas

Para dar formato a un elemento <table>, Bootstrap 5 nos ofrece las clases *.table*, *.table-bordered*, *.table-hover*, *.table-dark* y *.table-striped*

```
<table class="table table-striped">
  <thead>
    <tr>
      <th>Baraja española</th>
      <th>Baraja francesa</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Caballo</td>
      <td>Reina</td>
    </tr>
    <tr>
      <td>Rey</td>
      <td>Rey</td>
    </tr>
  </tbody>
</table>
```

http://ortuno.es/tablas.html

cards

Una *tarjeta* (*card*) es una caja redondeada dividida en cabecera, cuerpo y pie.

- Es útil para agrupar otros elementos como botones, formularios, imágenes, etc
- Sucesor de los antiguos *panels* en las versiones anteriores de Bootstrap
- Se les puede poner un color contextual de fondo añadiendo las clases que ya conocemos:
  .bg-primary, .bg-success, .bg-info, .bg-warning, .bg-danger, .bg-secondary, .bg-dark and .bg-light

```
<div class="card" style="width:400px">
  <div class="card-header">
    <h4 class="card-title">Gato Panchi</h4>
  </div>
  <div class="card-body">
    <img src="images/gato.jpg" alt="Fotografía de un gato
     llamado Panchi" width="300" class="mx-auto d-block">
  </div>
  <div class="card-footer">
    <a href="#" class="btn btn-primary float-end">Más
    ↪  información</a>
  </div>
</div>
```

http://ortuno.es/card.html

Formularios

Bootstrap incluye clases para mejorar el aspecto y usabilidad de los formularios

- El uso de `<label>` es necesario, no es válido escribir texto HTML para identificar los elementos del formulario
- Los distintos elementos de un formulario aparecen unos debajo de otros. Si los queremos unos al lado de otros, usaremos las filas y columnas de la rejilla

- A los `<label>` les añadimos `class="form-label"`
- A los elementos de entrada de texto, `<input>` y `<textarea>` les añadimos `class="form-control"`
- A los `<checkbox>` los metemos en un `<div>` al que añadimos `class="form-check"`
- A los `<input type="radio"> <input type="checkbox">` les añadimos `class="form-check-input"`

`http://ortuno.es/form_b5.html`

carousel

El componente `carousel` muestra fotografías que se desplazas horizontalmente, como un pase de diapositivas. Se les puede añadir título o cualquier otro texto

- El elemento de mayor nivel jerárquico del carrusel es un *div* con las clases *carousel* y *slide* Tiene un atributo *id* cuyo valor será referenciado por los botones que contenga

```
<div id="carrusel01" class="carousel slide"
 data-bs-ride="carousel">
```

El `.slide .carousel` contendrá tres *divs*

- `.carousel-indicators`
  Los puntos o pequeñas líneas que representan cada una de las fotos. Un *div* de clase *carousel-indicators*
- `.carousel-inner`
  Un *div* de clase *carousel-inner* con las imágenes.
  - Cada imagen es un *carousel-item*, que contiene la imagen y un *carousel-caption*. Importante: es muy recomendable que todas las imágenes tenga la misma relacion alto/ancho
- Los botones

`http://ortuno.es/carrusel.html`

Deshabilitar elementos

Como hemos visto, muchos elementos bootstrap admiten la clase *disabled* para indicar que tengan un aspecto gráfico distinto, deshabilitado

- Pero esto no los deshabilita realmente
- Para deshabilitar por completo un elemento, usamos el atributo *disabled*

```
<button type="button" class="btn btn-lg" disabled>Botón</button>

<input type="text" name="lname" disabled><br>
```

Depuración

Si la página no tiene el aspecto que buscas:

- Asegúrate de que todos los elementos están dentro de un *container*. Normalmente solo deberías usar uno para la página
- Usa el W3C validator. Detectará p.e. elementos sin cerrar (aunque no elementos cerrados en el sitio incorrecto)
- Comprueba que la estructura de los *div* está bien, que no has cerrado ninguno demasiado pronto o demasiado tarde. Un buen editor te ayudará con esto mostrando el código por niveles. P.e atom cuenta con los atajos Ctrl k Ctrl 1, Ctrl k Ctrl 2, Ctrl k Ctrl 3, etc
- Si usas Bootstrap, no añadas directamente reglas CSS. Excepto si estás seguro de lo que haces

Enlaces relacionados

- Documentación oficial

  `https://getbootstrap.com/docs/5.1/getting-started`

- Tutorial en w3schools

  `https://www.w3schools.com/bootstrap5/index.php`

JavaScript (i)

Introducción a JavaScript

JavaScript es un lenguaje de programación. Junto con HTML y CSS, es una de las principales tecnologías para presentar contenidos en la World Wide Web

- Creado por Brendan Eich, de Netscape, en 1995 como lenguaje de scripting para el navegador. Tardó 10 días en contruir el primer prototipo
- Está presente en el 100 % de los navegadores web modernos, donde no tiene rival

- El nombre *JavaScript* es poco afortunado. En su día se eligió por motivos de marqueting, para destacar que su sintaxis es similar a la de Java. Pero ahí acaba el parecido, es un lenguaje completamente distinto
- En 1996, Netscape encarga a Ecma International la normalización del lenguaje. La marca *java* pertenecía a Sun (luego a Oracle), así que el nombre formal del lenguaje se cambió a ECMAScript, aunque en la práctica lo normal es seguir llamándolo JavaScript

JavaScript Everywhere (1)

El éxito de internet lleva este lenguaje a ser masivamente utilizado, no solo en el navegador, se habla de *JavaScript everywhere*.
Aunque no fue inicialmente diseñado para esto, hoy puede usarse también en

- node.js
  Entorno de ejecución de JavaScript para el servidor.
- nw.js (antiguo node webkit)
  Electron (antiguo Atom Shell)
  Son entornos que permiten desarrollar aplicaciones nativas de escritorio mediante tecnologías web (JavaScript, html, css...)

JavaScript Everywhere (2)

- Mozilla Rhino. Implementación de JavaScript en java. Permite ejecutar código JavaScript fuera del navegador, en cualquier entorno donde esté disponible java
- Express.js
  Es un *Web Application Framework*, permite desarrollar aplicaciones web en el servidor. Basado en Node.js.
  Alternativa a Django o Ruby on Rails

Versiones de JavaScript (1)

- Brendan Eich crea JavaScript. 1995
- ECMAScript 1. 1997. Primera versión normalizada
- ECMAScript 2. 1998. Pequeños cambios
- ECMAScript 3. 1999
  do-while, regexp, excepciones, mejor tratamiento de cadenas (entre otros)
- ECMAScript 4.
  Abandonado en 2008, por falta de acuerdo sobre si las mejoras deberían ser más o menos drásticas

Versiones de JavaScript (2)

- ECMAScript 5. Año 2009. Modo strict, nuevos arrays, soporte JSON (entre otros)
- ECMAScript 6. Año 2015
  Cambios muy relevantes: módulos, orientación a objetos basada en clases, parámetros opcionales en funciones, variables locales a un bloque
  - En el año 2015 los navegadores en general no soportaban ECMAScript 6, era necesario *transpilar* el código a ECMAScript 5.
  - En la actualidad (año 2023) cualquier navegador medianamente actualizado lo soporta. Con alguna excepción, por ejemplo el uso de módulos. La necesidad del transpilador es cada vez menor

Críticas a JavaScript

Es frecuente hacer críticas negativas a JavaScript, por diferentes motivos, algunos justificados, otros no tanto

- No es un lenguaje especialmente elegante, sobre todo las primeras versiones. Fue diseñado apresuradamente y eso se nota. Pero ha ido mejorando mucho con el tiempo
  - En JavaScript moderno, si el programador usa la técnicas adecuadas, se puede generar código de gran calidad
- Los primeros intérpretes eran lentos. Esto también ha mejorado mucho. Incluso hay subconjuntos estáticos de JavaScript como asm.js cuyos programas pueden ejecutarse al 70 % de la velocidad del código compilado en C++
  - Esto es muy adecuado para algoritmos que verdaderamente lo necesiten

- Todos los números son del mismo tipo: float
- La distinción entre los tipos *undefined* y *null* es bastante arbitraria
- Hasta ECMAScript 3 no tenía excepciones. Los programas fallaban silenciosamente
- Hasta ECMAScript 6, no tenía variables limitadas a un bloque, solo globales o limitadas a la función
- Hasta ECMAScript 6, no tenía soporte (nativo) para módulos
- Los números se representan como Binary Floating Point Arithmetic (IEEE 754). Esto tiene sus ventajas para trabajar con binarios, pero representa muy mal las fracciones decimales

```
> 0.3===0.3
true
> 0.1+0.2===0.3
false
> 0.3-(0.1+0.2)
-5.551115123125783e-17
```

- La barrera de entrada para empezar a programar en JavaScript es baja. Como *cualquiera puede programar en JavaScript*, el resultado es que *en JavaScript acaba programando cualquiera*. Esto es, hay mucho código de mala calidad
- Es orientado a objetos. Pero en las versiones anteriores a ECMAScript 6, solo admitía orientación a objetos basada en *prototipos*. Este modelo es distinto al de lenguajes como C++ o Java, que están basados en clases y herencia. Si el programador fuerza al lenguaje a un uso como el de C++ o Java, el resultado es antinatural, incómodo y problemático. *It's not a bug, it's a feature*
- ECMASCript 6 admite programación orientada a objetos basada en prototipos y programación orientada a objetos basada en clases

Características de JavaScript

- Muy integrado con internet y el web
- La práctica totalidad de las herramientas necesarias para su uso son software libre
- El lenguaje no especifica si es interpretado o compilado, eso depende de la implementación
- Por motivos de seguridad, un programa JavaScript que se ejecute dentro del navegador (su diseño inicial), es imposible que
  - Acceda al sistema de ficheros
  - Envíe o reciba mensajes de propósito general por la red [4]

[4] solo puede usar WebSockets para comunicarse con servidores especializados

- Técnicas modernas como la compilación JIT (*Just In Time*) y el uso de bytecodes hacen que la división entre compiladores e intérpretes resulta difusa
- Podemos considerarlo un híbrido. Los *script engines* (motores) de JavaScript modernos tienden a ser más compilados que las primeras versiones
- Se acerca más a un lenguaje interpretado: el motor necesita estar siempre presente, la compilación se hace en cada ejecución y siempre se distribuye el fuente y solo el fuente

- Es dinámico. Los objetos se crean sobre la marcha, sin definir una clase. A los objetos se les puede añadir propiedades en tiempo de ejecución
- Es dinámicamente tipado. El tipo de las variables y objetos puede cambiar en tiempo de ejecución
- Multiparadigma, admite los paradigmas de programación:
  - Imperativa
  - Funcional
  - Basada en eventos (*event-driven*)
  - Orientada a objetos basada en prototipos
  - Desde ECMAScript 6, orientada a objetos basada en clases (orientación a objetos *tradicional*)

Holamundo

JavaScript no tiene una forma nativa de mostrar texto, emplea distintos objetos, dependiendo de en qué entorno se ejecute

- En el navegador puede escribir HTML mediante `document.write()`
  Esta forma es obsoleta, no debemos usarla
- Puede usar `console.log()`
  - En el navegador el texto saldrá por una consola (del propio navegador) [5].
  - En node.js, por la salida estándar
- Puede abrir una ventana con `window.alert()`

[5] Visible desde las herramientas de desarrollador. En Chrome, Firefox y otros el atajo es F12

Holamundo en HTML, incrustrado

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Hola, mundo</title>
   </head>
   <body>
      <script>
      document.write("Hola, mundo");
      </script>
   </body>
</html>
```

El elemento `<script>` puede aparecer 1 o más veces, tanto en la cabecera como en el cuerpo del documento HTML

`http://ortuno.es/holamundo01.html`

Holamundo en HTML, fichero externo

```html
<!DOCTYPE html>
<html>
   <head>
      <meta charset="utf-8">
      <title>Hola, mundo</title>
      <script src="js/holamundo.js"></script>
   </head>
   <body>
   </body>
</html>
```

holamundo.js:

```
console.log("Hola, mundo");
```

http://ortuno.es/holamundo02.html

Si la codificación del script es diferente de la codificación del fichero HTML, se indica con un atributo `charset` en el elemento `<script>`

Observa que normalmente indicaremos el *path* (trayecto) del fichero javascript de manera relativa. El trayecto no empieza por el carácter *barra* (/)

- `holamundo.js`

  En el mismo directorio donde está este html, hay un fichero llamando *holamundo.js*

- `js/holamundo.js`

  En el mismo directorio donde está este html, hay un directorio llamado *js*, y dentro, un fichero llamando *holamundo.js*

- `../practica03/holamundo.js`

  El directorio padre del directorio donde está este html, hay un directorio llamado *practica03*, y dentro, un fichero llamado *holamundo.js*

Si anteponemos la barra, el significado cambia por completo

- `/holamundo.js`

  En el directorio raiz, el fichero *holamundo.js*

- `/js/holamundo.js`

  En el directorio raiz, el subdirectorio llamado *js*, y dentro, un fichero llamando *holamundo.js*

- `/practica03/holamundo.js`

  El directorio raiz, el directorio *practica03*, y dentro, un fichero llamado *holamundo.js*

El directorio raiz será:

- Si el fichero lo leemos localmente, el directorio raiz de nuestro sistema de ficheros (disco duro)
- Si el fichero se exporta via web, el directorio raiz establecido por el servidor web,

Naturalmente, estas mismas ideas sobre los trayectos relativos y absolutos, son aplicables en cualquier lenguaje de programación y a cualquier fichero: una imagen jpg, una librería, etc

- Observa que un *path* que incluya la dirección absoluta del usuario casi siempre es un error muy severo, porque deja de funcionar en cuanto cambia el usuario o la máquina

```
/home/alumnos/jperez/js/holamundo.js
```

Podremos componer este trayecto de forma dinámica leyendo la variable de entorno *home*, pero nunca escribiendo la cadena literal

Holamundo mínimo aceptado

Apurando la norma de HTML, pueden incluso omitirse los elementos <html>, <body> y <head>. Se consideran entonces sobreentendidos, el siguiente ejemplo también sería válido, aunque no recomendable en absoluto

```
<!DOCTYPE html>
  <meta charset="utf-8">
  <title>Hola, mundo</title>
  <script  src="js/holamundo.js"></script>
```

http://ortuno.es/holamundo03.html

node.js

- El entorno Node.js permite usar JavaScript como un lenguaje de programación en el servidor o en la consola
- También es útil para desarrollar código que luego vaya a ejecutarse en el navegador

¿Donde colocar el código?

- Nunca es recomendable incrustar el JavaScript dentro del HTML, excepto tal vez para páginas muy sencillas
- Un defecto muy habitual es organizar el código de la lógica de negocio en función de las *pantallas*, (aunque sea en un fichero externo)
- O peor aún: repartirlo por los botones y formularios

Sugerencia: desarrolla la lógica de negocio en Node.js, luego intégralo en el HTML

- Excepto tal vez cosas muy sencillas

¿nodejs o node?

El intérprete de Node.js en principio se llama `node`

- En Linux
  - Este nombre ya estaba ocupado por otro programa. Así que las distribuciones Linux lo renombran a `nodejs`
  - Si el *otro* node no está instalado, normalmente `/usr/bin/node` es un enlace a `/usr/bin/nodejs`
    Por tanto, podemos usar indistintamente cualquiera de las dos formas
  - En resumen: según esté configurado nuestro Linux, el intérprete será `node`, `nodejs` o ambos indistintamente
- En macOS
  Generalmente se mantiene el nombre `node`

Entorno Linux

- Instalación

  ```
  apt-get install nodejs
  ```

- Ejecución

  ```
  nodejs holamundo.js
  ```

  O bien

  ```
  node holamundo.js
  ```

- Si el script es solamente para el servidor o el terminal y no para el navegador, también podemos ejecutarlo directamente

  ```
  jperez@alpha:~$ ./holamundo.js
  ```

  Para ello escribimos en la primera línea el siguiente comentario

  ```
  #!/usr/bin/nodejs
  ```

  O bien

  ```
  #!/usr/bin/env nodejs
  ```

Entorno macOS

- Podemos ejecutar node y pasarle como primer argumento el nombre del script

```
node holamundo.js
```

- O podemos añadir la siguiente primera línea al script

```
#!/usr/bin/env nodejs
```

Entorno del Navegador

- El código que vaya a ejecutarse en el navegador no puede empezar por `#!/usr/....`
  (la almohadilla normalmente no significa comentario en JavaScript)

Comentarios

Los comentarios se pueden indicar de dos formas

- `//Comentarios de una sola línea, con dos barras`

- ```
  /* Comentarios con barra y asterisco.
     Pueden ocupar varias líneas, pero no anidarse */
  ```
  ```

Sentencias y expresiones

En JavaScript hay

- Sentencias. *Hacen cosas*

  ```
  x = x+1;
  ```

  Un programa en JavaScript podemos considerarlo como una secuencia de sentencias (*statements*)

- Expresiones. *Devuelven valores*

  ```
  x + 1
  ```

  En cualquier lugar donde JavaScript espera una sentencia, también puede haber una expresión. Se denomina entonces *sentencia expresión*, (*expression statement*)

  - Con tal de que la expresión no empiece ni por llave ni por la palabra reservada *function*, porque esto provocaría ambiguedad con objetos y funciones
  - Donde se espera una expresión, no puede ir una sentencia

Uso del punto y coma

Un bloque es una secuencia de sentencias, entre llaves (`{}`)

- Las sentencias acaban en punto y coma
- Excepto las sentencias que acaban en un bloque
  - En este caso también se puede añadir un punto y coma, que se considera una sentencia vacia
- Si el programador no incluye los puntos y coma, el parser los añade con la *automatic semicolon insertion*. De hecho el JavaScript moderno tiende a omitir los puntos y coma, lo que en ciertos casos puede producir errores y confusiones.

Aquí recomendamos incluir siempre punto y coma al final de cada sentencia

use strict

En ECMAScript 5 aparece el *modo estricto*
Consiste en una serie de restricciones que producen un código de más calidad, menos propenso a errores. En general debemos usarlo siempre, para ello basta poner como primera sentencia del script

```
'use strict'
```

Es una cadena, no una sentencia. Aparece entre comillas.
Si tenemos que mezclar nuestro código con código antiguo, incompatible con el modo estricto, entonces podemos aplicar este modo función a función

```
function f(){
    'use strict'
    ...
}
```

Requisitos del modo estricto

Las principales normas de este modo son:

- Es necesario declarar explícitamente todas las variables
- Las funciones se deben declarar en *top level* o como mucho con un nivel de anidamiento (una función dentro de otra función). Pero no se admiten niveles más profundos de anidamiento.
- No se puede repetir el nombre un parámetro en la misma función
- El intento de modificar propiedades inmutables genera una excepción
- No se permite el uso de la sentencia *with*
- Un número que comienza por 0 no se considera que es un número octal

Tipos de datos

En JavaScript hay dos tipos de valores

- *primitive values*:
  boolean, number, string, null, undefined
- Objetos
  Los principales son: *plain objects*, arrays, regexp

Booleanos

- `true`
  `false`

Números

- A diferencia de la mayoría de los lenguajes de programación, solo hay un tipo para todos los números, incluyendo enteros y reales
- Hay dos *números* especiales: *Infinity* y *NaN* (*Not a Number*)

```
> 2/0
Infinity
> 0/0
NaN
> typeof(Infinity)
'number'
> typeof(NaN)
'number'
```

Strings (cadenas)

- Se puede usar la comila doble o la simple indistintamente (obviamente la comilla de apertura debe coincidir con la de cierre)

```
'lorem'  "ipsum"
```

  Ya que HTML usa la comilla doble, es más habitual usar en JavaScript la comilla simple
  Excepto en JSON donde es obligatorio usar la comilla doble

En JavaScript hay dos tipos de datos para indicar que falta información

- *undefined*
  - Una variable no ha sido inicializada
  - Se ha llamado a una función sin especificar algún parametro
  - Se intenta leer una propiedad no existente de un objeto
- *null*

  Es un objeto que no tiene valor. Más o menos podríamos decir que es un objeto vacío (auque el verdadero objeto vacío es `{}`)

```
'use strict'
// Variable no declarada
console.log(nombre)  // undefined

// Variable declarada
var nombre;
nombre = 'Juan';
console.log(nombre);  // juan

function saludo(n){
    console.log('Hola, ',n);
}

// Parámetro especificado
saludo('María');  // Hola, María

// Parámetro no especificado
saludo();  // Hola, undefined
```

```
var alumno = {};   // Declaramos objeto vacío
alumno.nombre = 'Pedro';
alumno.apellido1 = 'Pérez';

// Propiedad del objeto que existe
console.log(alumno.apellido1); // Pérez

// Propiedad del objeto que no existe
console.log(alumno.ape1);  // Undefined:
```

- La sentencia `console.log(nombre)` muestra *undefined* a pesar de que *nombre* no está declarada, incluso con el modo estricto. No salta una excepción porque sí declaramos la variable, en algún momento, más tarde.
- Si declaramos las variables con *let* y no con *var*, el comportamiento es algo mejor, saltará una excepción si intentamos usar una variable antes de definirla.
- El problema con las propiedades inexistentes del objeto, y el de los parámetros inexistentes, no tiene remedio. Ni con let ni con el modo estricto.

La distinción entre undefined y null es algo arbitraria, en ocasiones puede aparecer cualquiera de los dos, así que es normal escribir cosas como

```
if (x===undefined || x===null){
}
```

Esto equivale a

```
if (!x) {
}
```

Aunque es menos claro, porque hay otros valores que también son considerados *false* (false, 0, NaN y la cadena vacía)

- El objeto vacío {} y el array vacío [] se consideran *cierto*

```
> var x
undefined
> typeof(x)
'undefined'
> x=null
null
> typeof(x)
'object'
```

Sería más razonable que el tipo de null fuera undefined, pero la primera implementación de JavaScript hacía esto (por error) y luego ya se tomó como norma

Conversión de tipos

- La función global `Number()` convierte una cadena en número. Devuelve `NaN` en caso de error
- La función global `String()` convierte un número en cadena

```
'use strict'

let x,y;
x=Number("  3 ");
console.log(x,typeof(x));  // 3  'number'
y=String(x);
console.log(y,typeof(y));  // 3  string

console.log(Number("23j"));  // NaN
```

- Para comprobar los tipos disponemos de la función `typeof()`, que devuelve una cadena indicando el tipo de su argumento

Para comprobar si un objeto es `NaN`, no se puede usar directamente el comparador de igualdad, es necesaria la función *isNaN()*

```
'use strict'

let x = Number("xyz");
console.log(x);              // NaN
console.log(x === NaN);      // false

x = NaN;
console.log(x === NaN);      // false

console.log(isNaN(x));       // true
```

Otra de las paradojas de NaN es que no es igual a sí mismo

```
> NaN === NaN
false
```

Identificadores

Símbolos que nombran entidades del lenguaje: nombres de variables, de funciones, etc

- Deben empezar por letra unicode, barra baja o dólar. El segundo caracter y posteriores pueden ser cualquier carácter unicode
- Aunque los caracteres internacionales como eñes y tildes son fuentes potenciales de problemas: falta de soporte en el teclado del desarrollador, configuración del idioma en el sistema operativo, etc
- ¿Sensible a mayúsculas?
  - JavaScript: Sí
  - HTML: No
  - CSS: Sí

En entornos profesionales reales, el código fuente (identificadores, comentarios, etc) siembre debe estar en inglés. Obviamente el interfaz de usuario estará en español o en cualquier otro idioma conveniente para el usuario

- Sin embargo, en esta asignatura no escribiremos en inglés. La ventaja de un identificador en español, cuando estamos aprendiendo, es que queda claro que no es parte del lenguaje ni de ninguna librería estándar

Identificadores válidos:

- α `//Correcto, pero no recomendable`
  `alpha`
  `contraseña //Discutible`
  `$f //Discutible`
  `_valor`
  `x5`

Identificadores incorrectos:

- `5x`
  `#x`

Palabras reservadas

Las siguientes palabras tienen un significado especial y no son válidas como identificador:

```
abstract    arguments   await   boolean
break   byte    case    catch
char    class   const   continue
debugger    default delete  do
double  else    enum    eval
export  extends false   final
finally float   for function
goto    if  implements  import*
in  instanceof  int interface
let long    native  new
null    package private protected
public  return  short   static
super   switch  synchronized    this
throw   throws  transient   true
try typeof  var void
volatile    while   with    yield
```

Tampoco son identificadores válidos

```
Infinity NaN undefined
```

Números especiales

JavaScript define algunos valores numéricos especiales: `NaN` (Not a number), `Infinity`, `-Infinity`

```
'use strict'
let x,y;
x=1/0;
y= -1/0;
console.log(x);   // Infinity
console.log(y);   // -Infinity
console.log(typeof(x));    // number
console.log(typeof(y));    // number
console.log(typeof(NaN));  // number
```

Paradójicamente, `NaN` es un *number*

Operadores

Los principales operadores son

- Operadores aritméticos
  `+ - * / % ++ --`
- Operadores de asignación
  `= += -=`
- Operadores de cadenas
  `+ +=`

```
'use strict'
let x;
x=0;
++x;
console.log(x); // 1
x+=2;
console.log(x); // 3
--x;
console.log(x); // 2
x-=2;
console.log(x); // 0

x='hola'+'mundo';
console.log(x); // 'holamundo'
x+="!";
console.log(x); // 'holamundo!'
```

- JavaScript 1.0 solo incluía el *lenient equality operator*, comparador de igualdad tolerante

  == !=

  Hace conversión automática de tipos.

```
> '4'==4
true
```

  Esto produce muchos resultados problemáticos

```
> 0==false
true
> 1==true
true
> 2==false
false
> 2==true
false
> ''==0
true
> '\t123\n'==123
true
> 'true'==true
false
```

- Comparador de igualdad estricto.
  Aparece en JavaScript 1.3, es el que deberíamos usar siempre
  `===` `!==`
- Mayor y menor
  `>` `<` `>=` `<=`
- Operador condicional
  `condición? valor_si_cierto : valor_si_falso`

```
> edad=18
18
> (edad>17)? "mayor_de_edad":"menor"
'mayor_de_edad'
```

- Operadores lógicos
  `&&` `||` `!`

```
> !(true && false)
true
```

Funciones

Una función es una secuencia de instrucciones empaquetada como una unidad. Acepta 0 o más valores y devuelve 1 valor.
Las funciones en JavaScript pueden cumplir tres papeles distintos

- *Nonmethod functions*. Funciones *normales*, no son una propiedad de un objeto. Por convenio, su nombre empieza por letra minúscula
- *Constructor*. Sirven para crear objetos. Se invocan con el constructor `new`.
  Por convenio, su nombre empiezan por letra mayúscula
  `new Cliente()`
- Métodos. Funciones almacenadas como propiedad de un objeto

Declaración de funciones

Hay cuatro formas de declarar una función

- Mediante una declaración. Es la forma más habitual

```
function suma(x,y){
    return x+y;
}
```

- Mediante una expresión. Función anónima. Habitual por ejemplo en JQuery

```
function(x,y){
    return x+y;
}
```

- Mediante *funciones flecha* (*arrow functions*)
  Es una sintaxis compacta para definir funciones anónimas
  ```
  x => x + 1;
  ```
- Mediante el constructor `Function()`. Crea la función a partir de una cadena. No recomendable.
  ```
  new Function('x','y','return x+y');
  ```

Función Flecha

La declaración tradicional de una función incluye nombre

```
function f(x){
  return x + 1;
}
```

Eliminamos el nombre y tenemos las funciones anónimas

```
function (x){
  return x + 1;
}
```

Eliminando la palabra *function* y ponemos una flecha entre los argumentos y la llave de apertura

```
(x) => {
  return x + 100;
}
```

Eliminamos las llaves y la palabra *return* (el return es implícito)

```
(a) => a + 100;
```

Finalmente, eliminamos los paréntesis del argumento y tenemos la sintaxis de las funciones flecha

```
a => a + 100;
```

Si no hay argumentos, o hay más de uno, entonces los paréntesis son imprescindible

```
(x, y) => x + y + 1;
```

```
() =>  1;
```

Si el cuerpo tiene más de una línea, es necesario añadir tanto las llaves como la palabra *return*

```
(a, b) => {
  let margen = 0.1;
  return a + b + margen;
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Funciones de primera clase

JavaScript es un lenguaje con funciones *de primera clase*. Esto significa que el parámetro de una función, además de booleano, número, cadena, etc, también pueden ser otra función

- Podemos declarar una función con nombre y luego usar su nombre como parámetro. Para el programador nuevo en JavaScript esto es lo más claro, pero no es idiomático
- En vez del nombre, podemos escribir la definición de una función anónima
- O podemos emplear la notación flecha

Veremos a continuación tres ejemplos, equivalentes

Función con nombre como argumento

```
'use strict'

function f(revisa, x){
    if (revisa(x)) {
    console.log('Haz esto y lo otro con',x);
    } else {
    console.log('No hagas nada con',x);
   }
}

function revision_a(x){
    const Umbral = 12.3;
    return (x >= Umbral);
}

function revision_b(x){
    return (x[0] != '_');
}

if( revision_a, 17);   // Haz esto y lo otro con 17
if( revision_b, "_hola");  // No hagas nada con _hola
```

Importante: Pasamos como argumento el nombre de la función, sin paréntesis. Ejemplo:

```
if( revision_a, 17);
```

Se escribiéramos paréntesis, estaríamos pasando el resultado de llamar a la función, no la función en si misma. Ejemplo:

```
if( revision_a(), 17);      //  ¡MAL!
```

Función anónima como argumento

```
'use strict'

function f(revisa, x){
    if (revisa(x)) {
    console.log('Haz esto y lo otro con ',x);
    } else {
    console.log('No hagas nada con ',x);
   }
}

f( function(x){
        const Umbral =12.3;
        return (x >= Umbral);
    } , 17);              // Haz esto y lo otro con 17

f( function(x) {
        return (x[0] != '_');
    } , "_hola");         // No hagas nada con _hola
```

Función flecha como argumento

```
'use strict'

function f(revisa, x){
    if (revisa(x)) {
    console.log('Haz esto y lo otro con ',x);
    } else {
    console.log('No hagas nada con ',x);
   }
}

const Umbral = 12.3;

f( x => x >= Umbral , 17);     // Haz esto y lo otro con 18

f( x => x[0] != '_',  "_hola");   // No hagas nada con _hola
```

Hoisting

JavaScript hace *hoisting* (elevación) con las funciones.
El motor de JavaScript mueve las declaración al principio del bloque,

```
'use strict'
console.log(f(1));   //2

function f(x){
   return x+1;
}
```

Paso por valor

En JavaScript, el paso de parámetros a una función es por valor (por copia). La función recibe una copia del valor del argumento. Si la función modifica este valor, el argumento original no queda modificado

```
'use strict'
function f(x){
    console.log(x)  // 3
    x = x + 1;
    console.log(x)  // 4
}
var a=3;
console.log(a);  // 3
f(a);
console.log(a);  // 3
```

Paso por referencia

En JavaScript no existe el paso por referencia. Si realmente lo necesitamos, podemos simularlo envolviendo el valor en un array

```
'use strict'
function f(a){
    a[0] = a[0] + 1;
    console.log(a[0]);  // 4
}
let x = 3;
let a = [x];
console.log(a);  // [3]
f(a);
console.log(a);  // [4]
```

Valor devuelto

Una función siempre devuelve exactamente 1 valor. En caso de que la función no incluya la sentencia `return`, el valor es `undefined`

```
'use strict'

function f(){
}
console.log(f());   // undefined


function g(){
    console.log('hola');
}
console.log(g());   // undefined
```

Número de parámetros

Muchos lenguajes de programación obligan a que el número de parámetros en la declaración de una función sea igual al número de argumentos cuando se invoca.
JavaScript, no. Si faltan argumentos, se consideran `undefined` y si *sobran* se ignoran

```
'use strict'
function f(x,y){
    return x+y;
};
console.log(f(1));    // NaN
console.log(f(2,2));  // 4
console.log(f(1,1,1)); // 2
```

Podemos conocer el número de argumentos recibidos usando la propiedad *length* del objeto predefinido *arguments*

```
'use strict'

function f(x,y){
    console.log("Argumentos recibidos:",arguments.length);
}

f('a');           // 1
f('a','b');       // 2
f('a','b', 'c');  // 3
```

Valores por omisión

Para dar un valor por omisión a un parámetro omitido en la invocación de una función, podríamos hacer lo siguiente

```
'use strict'

function f(x){
    if (x===undefined) {
        x=0};
     return x + 1 ;
};
console.log(f()); //1
```

Aunque la forma habitual en JavaScript 5 y anteriores era esta otra:

```
'use strict'

function f(x){
    x = x || 10;      //  línea 4
    return x ;
};

console.log(f(7));    //  7
console.log(f());     // 10
```

Línea 4. El operador *or* evalúa *en cortocircuito*

- Si x está definido, se considera *cierto* y la expresión devuelve x
- Si no lo está, se considera *falso* y la expresión devuelve el segundo valor

Aunque es una solución habitual, no es elegante

- Si pasamos explícitamente el valor *false*, la función devolverá el valor por omisión
- Si pasamos un valor, se devuelve ese valor (el or lo considera cierto) aunque para javascript no sea ni *true* ni *false*

```
'use strict'

function f(x){
    x = x || 10;
    return x ;
};

console.log(f(false));  // 10
console.log(f(1));      //  1

console.log(1 === true);    // false
console.log(1 === false);   // false
```

En general deberíamos evitar construcciones rebuscadas y poco claras. Pero este caso concreto podemos considerarlo idiomático en JavaScript, resulta aceptable

En ECMAScript 6 es mucho más sencillo

```
'use strict'

function f(x=10){
    return x;
}
console.log(f(5));      //     5
console.log(f());       //    10
console.log(f(false)); // false
```

Los motores actuales (año 2023) suelen tener esto implementado.
Si son un poco antiguos, no

Ámbito de las variables

Ámbito (*scope*)
Zona del código donde una variable es accesible
Hay tres tipos de variables

- Locales, declaradas con `var`. O declaradas implícitamente (si no usamos el modo estricto)
- Locales, declaradas con `let`. Aparecen en ECMAScript 6
- Globales
  Declaradas fuera de una función. En ECMAScript 6 se pueden declarar con `var` o con `let`. Hay una pequeña diferencia entre ambas que hace preferible `let`

`http://2ality.com/2015/02/es6-scoping.html#the-global-object`

Variables Globales

Son variables accesibles desde todo el script
En el caso de JavaScript incrustado en HTML, todos los scripts incluidos en la misma página HTML comparten el objeto Window y por tanto, las variables globales

- Algunas metodologías recomiendan que las variables globales se usen lo menos posible
- Otras, que no se usen nunca

```
'use strict'
let x=0;  // Global por declararse fuera de función
function f(){
  x=3;  // Modifica la variable global
}

function g(){
  return(x);
}

f();
console.log(g());  //3
```

Variables locales con var

Las variables declaradas con `var` son locales a su función

```
'use strict'
function f(){
  var x=0; // Variable local de f
  g();
  console.log(x);  //0. No le afecta el cambio en g
}

function g(){
  var x=3; // Variable local de g
}

f();
```

El modo estricto obliga a declarar las variables, pero `var` permite usar primero y declarar después. Incluso declarar dos veces

```
'use strict'
// Variable no declarada (todavía)
console.log(nombre)  // undefined

// Variable declarada
var nombre = 'Juan';
console.log(nombre);  // Juan

// Variable declarada dos veces
var nombre = 'María';
console.log(nombre);  // María
```

Variables locales con let

Las variables declaradas con `let`

- Son locales a su bloque
- Tienen el comportamiento habitual en la mayoría de los lenguajes de programación
  - La misma variable no se puede declarar dos veces
  - Es necesario declarar primero, usar después
- Aquí recomendamos usar siempre `let`, a menos que tengamos que programar en una versión antigua de JavaScript

```
'use strict'
function f() {
  var x = 1;
  if (true) {
    var x = 2;  // La misma variable. Destruimos valor previo
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function g() {
  let x = 1;
  if (true) {
    let x = 2;  // Variable diferente, local del bloque
    console.log(x);  // 2
  }
  console.log(x);  // 1. Mantiene el valor previo
}
f();
g();
```

En ECMAScript 5 y precedentes, para conseguir algo similar a esto se usaba un truco no muy elegante denominado IIFE (immediately invoked function expression)

- Consiste en declarar una función sin nombre, abrir y cerrar paréntesis a continuación para que se invoque inmediatamente y ponerlo todo entre paréntesis

```
(funtion() {
    }());
```

Constantes

Como en muchos otros lenguajes, podemos declarar constantes usando *const*. Equivale a declarar con *let*, solo que el objeto no podrá ser reasignado

```
'use strict'

const a = 5;
a = 4  ;   // ¡MAL! TypeError
```

Este programa generará una excepción en su última línea

Atención, declarar un objeto con *const* hace que no se pueda reasignar, pero no significa que sea inmutable. Podremos cambiar sus propiedades[6]

```
'use strict'

const b={
    x:"lorem",
    y:"ipsum"
}

//b = {}  //  Esto generaría un TypeError

b.x = "otra cosa";  // Esto es correcto
console.log(b.x);  //  Escribe "otra cosa"
```

[6] Este ejemplo usa un plain object, que veremos al final de este tema

Condicional

La sentencia `if` funciona como en muchos otros lenguajes de programación

```
'use strict'
var x="ejemplo";
if (x.length < 4){
    console.log("Cadena muy corta");
};

if (2 > 0) {
    console.log("cierto");
}
else {
    console.log("falso");
};
```

Es recomendable usar siempre los bloques de sentencias (las llaves). Aunque si la sentencia es única, pueden omitirse

```
if (2 > 0) console.log("cierto");
else console.log("falso");
```

switch

Evalúa la expresión entre paréntesis después de `switch` y salta a la cláusula `case` cuyo valor coincida con la expresión. O a la cláusula `default` si ninguna coincide.

```
'use strict'

let y;
let x=":";
switch(x){
    case(';'):
        y="punto y coma";
        break;
    case(':'):
        y="dos puntos";
        break;
    default:
        y="caracter desconocido";
}
console.log(y);  // dos puntos
```

Después de cada `case` se indican una o más sentencias, lo habitual es que la última de ellas sea `break`

- También se puede concluir lanzando una excepción con `throw` o saliendo de la función `return`, aunque esto último no es recomendable

Si no se incluye ninguna sentencia de finalización, la ejecución continúa.

- Si esa es la intención del programador (y no un olvido), es recomendable indicarlo de forma explícita
- Tradicionalmente se usa la expresión `fall through` (*cae a través, pasa, se cuela*)

```
'use strict'

let x='ubuntu';
let so="";
switch(x){
    case('ubuntu'):
        //fall through
    case('debian'):
        //fall through
    case('fedora'):
        //fall through
    case('redhat'):
        so='linux';
        break;
    case('macos'):
        so="macos"
        break;
    default:
        so='no soportado';
}

console.log(so);
```

La expresión de cada case puede ser cualquiera:

```
'use strict'
function cuadrante(x,y){
    let r;
    switch(true){
        case( x>= 0 && y>=0):
            r=1;
            break;
        case( x< 0 && y>=0):
            r=2;
            break;
        case( x< 0 && y<0):
            r=3;
            break;
        case( x>= 0 && y<0):
            r=4;
            break;
        default:
            r=NaN;
    }
    return r;
}
console.log(cuadrante(1,-1));  // 4
```

while

```
'use strict'

let x=5;
let cadena="";

while(x>0){
    --x;
    cadena+="*";
}
console.log(cadena);  //*****


x=5;
cadena="";

while(true){
    if(x<1) break;
    --x;
    cadena+="*"
}
console.log(cadena);  //*****
```

for

La sentencia `for` también es como en C y muchos otros lenguajes

- Entre paréntesis y separado por punto y coma se indica la sentencia inicial, la condición de permanencia y la sentencia que se ejecuta después de cada ejecución del cuerpo
- A continuación, el bloque (o sentencia) a ejecutar

```
'use strict'
let cadena="";
for(let i=0; i<5; ++i){
    cadena+="*";
}
console.log(cadena);  //*****
```

Bucles sobre cadenas

- Podemos acceder a los caracteres individuales de una cadena mediante corchetes
- La primera posición es la 0
- La longitud de la cadena se puede consultar con la propiedad `length` de la cadena

```
'use strict'
let x;

x="Lorem Ipsum";

for (let i=0; i<x.length; ++i){
    console.log(x[i]);
}
```

Como hemos visto, JavaScript tiene una característica peligrosa: si intentamos acceder a una propiedad inexistente de un objeto, simplemente obtenemos `undefined`
Supongamos que, por error, escribamos `x.lengh` en vez de `x.length`

```
for (let i=0; i<x.lengh; ++i){   //¡ERROR! Debería ser length, no lengh
    console.log(x[i]);
}
```

- En la primera iteración, la condición del bucle será `0 < undefined`
- Esto se evalúa como `false`
- El bucle concluye silenciosamente, sin generar ningún error

Generalmente esto es un comportamiento no deseado, puede resultar un error difícil de trazar

En ECMAScript 6 podemos recorrer una cadena de forma muy conveniente con for-of

```
'use strict'
let x="Lorem Ipsum";

for (let c of x){
    console.log(c);
};
```

Manipulación de cadenas

Las cadenas tienen diversos métodos que permiten su manipulación
Todos estos métodos devuelven una nueva cadena, dejando la original intacta

- `toUpperCase()` y `toLowerCase()` devuelven la cadena en mayúsculas/minúsculas

```
> 'contraseña'.toUpperCase()
'CONTRASEÑA'
> 'LoReM IPsum'.toLowerCase()
'lorem ipsum'
```

- El método `trim()` devuelve la cadena eliminando los espacios a la izquierda y a la derecha
  - Espacios en sentido amplio, incluye tabuladores y el caracter fin de línea

```
> '    ABC  '.trim()
'ABC'
```

- El método `indexOf()` devuelve la posición de la primera aparición de una subcadena. O el valor -1 si no está incluida

```
> '__abc'.indexOf('abc')
2
> '__abc'.indexOf('xxx')
-1
```

- `lastIndexOf()` devuelve la última aparición de una subcadena. O el valor -1 si no está incluida

```
> 'a.tar.gz'.lastIndexOf('.')
5
```

- `slice(x,y)` devuelve la subcadena comprendida entre la posición `x` (incluida) y la `y` (excluida)

```
> '0123'.slice(0,3)
'012'
```

- Si `x` o `y` exceden las dimensiones de la cadena, no es problema

```
> 'abc'.slice(0,7)
'abc'
> 'abc'.slice(-5,7)
'abc'
```

- Si `x` es mayor o igual que `y` , devuelve la cadena vacía

```
> 'abc'.slice(3,2)
''
> 'abc'.slice(2,2)
''
```

- El método `split(c)` trocea una cadena, usando el caracter `c` como separador. Devuelve un array

```
> "a,b,c".split(',')
[ 'a', 'b', 'c' ]
```

- El método `replace(x,y)` devuelve una cadena donde la subcadena `x` ha sido reemplazada por `y`

```
> 'color beige'.replace('beige','crema')
'color crema'
```

Arrays

En JavaScript disponemos de un tipo de objeto denominado *array* (lista). Un array es un objeto donde se hace corresponder un número natural con un valor

- Se declara entre corchetes, en el interior habrá elementos, separados por comas
- A diferencia de otros lenguajes más sencillos como C y similares, los *arrays* de JavaScript
  - Son dinámicos: no es necesario fijar su tamaño a priori
  - Están formados por elementos que no necesariamente han de ser del mismo tipo

```
'use strict'

let a,b,c,d;

a=[ ];  // array vacío
b=[7, 8, 9];  // array con números
c=['rojo', 3, 0];  // array con elementos heterogéneos
d=[ [0, 1], [1, 1]];  // array con arrays anidados
```

Los arrays en JavaScript tienen muchos métodos disponibles. Mostramos algunos de los principales

- Atención: algunos son *destructivos*, esto es, modifican el array
- Otros, devuelven un array con la modificación requerida

```
'use strict'
let a,x;
a=['sota', 'caballo'];

// Longitud del array
console.log(a.length);  // 2

// Acceso a un elemento individual
console.log(a[1]);  // caballo
// JavaScript no admite índices negativos

// Añadir un elemento al final
a.push('rey');
console.log(a);   // [ 'sota', 'caballo', 'rey' ]

// Extraer un elemento al final
x=a.pop(); //
console.log(x); // rey
console.log(a);   // [ 'sota', 'caballo']
```

```
// Extraer un elemento al principio
x=a.shift();
console.log(x); // sota
console.log(a);   // [ 'caballo']

// Añadir un elemento al principio
a.unshift('alfil');
console.log(a); // ['alfil', 'caballo']

// Añadir un elemento, creando huecos
a[3]="torre";
console.log(a);   // ['alfil', 'caballo', , 'torre']

// La propiedad length incluye los huecos
console.log(a.length);  // 4

// Truncar un array
a.length=0;
console.log(a);   //   []

a=['alfil', 'caballo', 'torre']
a.reverse();
console.log(a);   //  ['torre', 'caballo', 'alfil']
```

slice

El método *slice()* devuelve una *rodaja* de una lista. No es destructivo

```
'use strict'
let a;
a=['sota', 'caballo', 'rey', 'as'];

let i = 1;
let j = 3;

console.log(a[i]) // caballo

// El método slice(i,j) devuelve la sublista
// entre el elemento i (incluido) y el j (excluido)
console.log(a.slice(i,j)) // ['caballo', 'rey']

// Si solo indicamos un valor, se entiende que la
// rodaja empieza ahí y acaba al final

//console.log(a.slice(,2));  // ERROR
console.log(a.slice(2));  // ['rey', 'as']
console.log(a.slice(2,));  // ['rey', 'as']
```

```
// Si i o j son negativos, cuentan desde el final,
// el -1 es el último
console.log(a.slice(-3,-1));  // ['caballo', 'rey']

// slice no es destructivo
console.log(a);   // ['sota', 'caballo', 'rey', 'as'];
```

splice

El método *splice()* devuelve una rodaja de una lista, de forma destructiva

```
'use strict'
let a=['sota', 'caballo', 'rey', 'as'];
let i = 1;
let n = 2;

// splice(i,n) devuelve n valores desde el i.
console.log(a.splice(i,n));   // ['caballo', 'rey'];

// splice es destructivo
console.log(a); // ['sota', 'as']

// Si i es negativo, cuenta desde el final,
// el -1 es el último
a=['sota', 'caballo', 'rey', 'as'];
i = -2
console.log(a.splice(i,1)); // rey
// Si n es negativo, develve lista vacía
console.log(a.splice(i,-2)); // []
```

Concatenar arrays

- Los arrays disponen del método `concat`, que permite concatenar ese array con otro(s) array(s)
- Es un método no destructivo: el array original permanece inalterado, devuelve un array nuevo

```
'use strict'
let a = ['alpha', 'beta'];
let b = ['gamma', 'delta'];
let c = a.concat(b);
console.log(c);   // [ 'alpha','beta','gamma','delta' ]

c = a.concat(a,b);  // Concateno 'a' consigo misma y con 'b'
console.log(c);  // [ 'alpha','beta','alpha','beta','gamma','delta' ]
```

Hay diversas formas de recorrer un array

- Al estilo C

```
'use strict'
let l=["a",,"c"];
for(let i=0; i<l.length; ++i){
    console.log(l[i]);
}
// a undefined c
```

También itera sobre los huecos

- Con el método `forEach`
  - Recibe una función, que se aplicará a cada elemento del array
  - Es habitual pasar una función anónima

```
'use strict'
let l=["a",,"c"]
l.forEach(function(x){
    console.log(x);
});
// a c
```

Se ignoran los huecos

- Especialmente conveniente es `for-of`, disponible en ECMAScript 6

```
'use strict'
let l=["a",,"c"]
for(let x of l){
    console.log(x);
}
// a undefined c
```

También itera sobre los huecos

No debemos usar for-in para recorrer un array, porque los arrays, además de índice, pueden tener otras propiedades que también se recorrerían

```
'use strict'
let a,x;
a=[7, 8];
a.color='azul'
for (x in a){
    console.log(x); // 0, 1, color
}
```

Los métodos `indexOf()` y `lastIndexOf()` se comportan de igual forma que sobre las cadenas

```
'use strict'
let a;
a=[7, 8, 9, 7];

console.log(a.indexOf(9)); // 2
console.log(a.indexOf(3)); // -1
console.log(a.lastIndexOf(7)); // 3
```

Plain Objects - Objetos literales

Además de los objetos *array*, en JavaScript disponemos de los *plain objects*[7], denominados en español *objetos literales*

- Análogos a los diccionarios de otros lenguajes
- Cada objeto está compuesto por un conjunto de propiedades
- Cada propiedad está formada por
  - Clave
    Una cadena
  - Valor
    Puede ser un valores primitivos (booleano, número, cadena, null, undefined) o bien una función o bien otro objeto

Su declaración

- Está delimitada entre llaves
- Clave y valor van separadas por el carácter *dos puntos*
- Las propiedades van separadas por comas. Desde JavaScript 5 se permite que la última propiedad también acabe en coma

[7] También llamados Plain Old JavaScript Object

```
'use strict'
let x={
    unidades:2,
    color:'verde',
    tamaño:'grande',
};
console.log(x); // { unidades: 2, color: 'verde', 'tamaño': 'grande' }
console.log(x.unidades);  // 2
console.log(x.precio); //undefined
```

Además de unar la notación
`objeto.clave`
se puede usar la notación
`objeto["clave"]`

- Tiene la ventaja de que permite usar claves calculadas

```
'use strict'
let p={ latitud:40.3355, longitud:-3.8773 };

console.log(p.latitud);  // 40.3355
console.log(p["latitud"]);  // 40.3355

let clave="latitud";
console.log(p[clave]);  // 40.3355
```

Podemos obtener la lista de claves de un objeto usando el método `keys()` del *built-in object* `Object`

`Object.keys(miObjeto)`

```
'use strict'
let p={ latitud:40.3355, longitud:-3.8773 };

let clave, claves;
claves=Object.keys(p);
console.log(claves); // [ 'latitud', 'longitud' ]

for (clave of claves){
    console.log(clave, ":", p[clave]);
};
/*
  latitud : 40.3355
  longitud : -3.8773
*/
```

Para saber si un objeto es un array, disponemos de la función `Array.isArray()`

```
'use strict'
let miObjeto={color:"verde"};
let miLista=[1,2];

console.log(typeof(miObjeto));  //object
console.log(typeof(miLista));  //object

console.log(Array.isArray(miObjeto));  //false
console.log(Array.isArray(miLista));   //true
```

Una función solo devuelve 1 argumento. Si necesitamos que devuelva más, podemos usar estos objetos

```
'use strict'
function f(x,y){
    let r={};
    r.suma=x+y;
    r.producto=x*y;
    return r;
};
console.log(f(2,3)); // { suma: 5, producto: 6 }
console.log(f(2,3).suma);    // 5
console.log(f(2,3).producto);  // 6
```

Referencias

- *Speaking JavaScript. An In-Depth Guide for Programmers*
  Axel Rauschmayer. O'Reilly Media, 2014
  `https://learning.oreilly.com/library/view/speaking-javascript/9781449365028`
- *JavaScript: The Definitive Guide, 7th Edition*
  David Flanagan. O'Reilly Media, 2020
  `https://learning.oreilly.com/library/view/javascript-the-definitive/9781491952016`

DOM: Document Objecto Model

DOM: Document Object Model

DOM, Document Object Model es un API que permite procesar una página HTML usando JavaScript. El documento HTML se represente en una estructura con forma de árbol

- Estándar de Internet, aparece en 1998, normalizado por el W3C
- Es el interfaz que emplean los navegadores web internamente
- Cuando un navegador carga una página HTML, la procesa para convertirla en la estructura del DOM. Desde ahí se representa en pantalla.
- Los cambios que se realicen en la página desde JavaScript se hacen directamente en el DOM. El HTML no se vuelve a utilizar

La forma de procesar el DOM ha ido cambiando con los años

- En la década de 2000 se usaba JavaScript *tal cual*, no había alternativa
- En la década de 2010 lo más habitual solía ser emplear la librería jQuery, resultaba más cómodo y conveniente
- JavaScript va mejorando, tomando las mejores ideas de jQuery. En 2020 hay una tendencia clara en el abandono de jQuery y en la vuelta al JavaScript *tal cual*. A veces se le llama *Vanilla JavaScript*
  - La palabra inglesa *vanilla*, literalmente significa *vainilla*. En sentido figurado significa *básico*, *sin adornos* o *convencional* [8]

[8] Este uso aparece en los años 1950, cuando los helados con sabor vainilla (artificial), se convierten en los más habituales por ser los más baratos

Normalización del DOM y el lenguaje JavaScript

Tanto el lenguaje como el DOM tienen un estándar que los navegadores suelen seguir bastante bien

- En principio, las prácticas de esta asignatura, como cualquier otro programa similar, se podrá ejecutar sin cambios
  - En cualquier navegador web, a menos que sea muy antiguo
  - En cualquier sistema operativo: Linux, Windows, macOS, Android, iOS, iPadOS

Si el resultado es diferente, normalmente se debe a que

- Hay algún error en el código, que distintas plataformas pueden tratar de forma distinta. Por eso es tan importante validar el código
- Usamos alguna característica muy reciente, aún no incluida en el navegador

Hay muchas técnicas en JavaScript (funciones, métodos, atributos, etc) que han quedado obsoletas

- Si están anticuadas no es por moda o capricho: son menos eficientes, menos seguras o más complejas
- Siguen funcionando, para mantener la compatibilidad del código antiguo
- Aunque *funcionen* no debemos usarlas
  - Excepto tal vez en el mantenimiento de código antiguo
- Debemos saber reconocerlas, para evitarlas. Debemos prestar atención a los libros, tutoriales o recetas antiguos
  - Todo es nuevo cuando se escribe, hasta que deja de serlo. Cualquier documento técnico debería incluir su fecha (normalmente basta el año o el mes y el año)

Modificar el HTML ¿Para qué?

Programando sobre el DOM se puede hacer prácticamente cualquier cosa con una página web

- Normalmente lo que deberíamos buscar es funcionalidad útil que mejore la experiencia de usuario
- Deberíamos evitar los efectos que llamen la atención gratuitamente, comportamiento no estándar o poco intuitivo, adornos que acaban molestando, etc

Funcionalidad que realmente mejora la experiencia de usuario:

- Es normal que una aplicación tenga muchos parámetros, difíciles de asimilar para el usuario
  Ocultar unos y mostrar otros facilita su trabajo
  - Se puede ocultar y/o marcar como deshabilitado lo que en cierto momento no se puede hacer
  - Jerarquizar el interfaz. Por ejemplo modo básico, modo normal, modo experto
- Ofrecer información y ayuda contextual
- Presentar la información en distintos formatos o unidades
- Validación de formularios

- Formularios mejorados
  Ejemplos
  - Una entrada donde el usuario indica un porcentaje desplazando una barra, no introduciendo un número
  - Una entrada que inmediatamente actualiza otra información. *Si gasta 20 entonces le quedan 80*
- Información sobre el progreso de lo que el usuario ha pedido. P.e *progress bar* en porcentaje, o en unidades de tiempo. O estimaciones del tiempo restante
- Información en tiempo real sobre sucesos diversos

- Generación de gráficos *bitmap*
  HTML Canvas.
- Generación de gráficos vectoriales.
  HTML SVG
  SVG: estándar para gráficos vectoriales. Muy extendido, soportado por ejemplo en aplicaciones como Adobe Illustrator o Inkscape
  Se pueden incrustar en el HTML y generar desde javascript.
  La librería más habitual es d3.js

  `https://github.com/d3/d3/wiki/Gallery`
- ...

Ejecución de un programa Javascript

La ejecución de un programa JavaScript en el navegador tiene dos partes

1. Ejecución secuencial
   Se carga el documento HTML y se ejecutan todos sus scripts, normalmente en el orden en que aparecen en el html [9]. Suele durar décimas de segundo
2. Ejecución asíncrona, dirigida por eventos
   El código empieza a responder a los eventos de usuario (y algunos otros: carga de ficheros, red y errores). Dura todo el tiempo que esté la página web en el navegador

En la primera fase de ejecución secuencial, el código que vaya a ocuparse de algún evento se registra, para que en la segunda fase, cada vez que aparezca el evento, se lance

[9] Aunque esto se puede alterar con async y defer

Correspondencia HTML - DOM

Todos los elementos de la página HTML que recibe el navegador se reproducen en una estructura análoga en el DOM

- Por cada elemento HTML hay un objeto *Element* en el DOM
- Por cada atributo del elemento HTML, hay una propiedad en el elemento JavaScript

Objetos Globales

En el DOM hay dos objetos globales muy importantes. Están predefinidos, siempre están presentes en cualquier documento.

1. *Window*
2. *document*

Pero además

- Cualquier otra constante, variable, función o clase que defina el programador, se comparte por todos los scripts y módulos de la misma ventana o pestaña.
  Esta idea es fundamental: aunque tengas varios ficheros `.js`, si están incluidos en el mismo `.html`, es como si fueran un único `.js`
- En otras palabras, todos los scripts de una ventana o pestaña comparten el mismo espacio de nombres

Objeto *Window*

- El código JavaScript que se ejecuta en el navegador tiene un objeto global llamado *Window*
- Hay uno por cada ventana (o pestaña) del navegador, compartido por todos los scripts y todos los módulos (pero no los *WebWorkers*) [10]

Objeto *document*

- Cuando el navegador procesa el HTML, crea un objeto *document*
- Es el objeto principal del DOM, en él se insertan todos los objetos *Element* y todos los nodos de texto

[10] En node.js este objeto se llama global. En los WebWorkers este objeto se llama WorkerGlobalScope

Eventos

Como hemos visto, una vez cargado y procesado el HTML, el programa JavaScript dentro del navegador entra en la fase dirigida por eventos

- En el contexto del desarrollo de software, un evento es una acción que sucede y que ha de ser tratada por el programa. Normalmente se genera de forma asíncrona y proviende de una fuente externa al propio programa (red, disco, ratón, teclado, reloj, etc)
- Uno de los tipos de eventos más importantes en cualquier programa suelen ser los eventos *de ratón*. Esta es la denominación habitual, pero informal. En rigor no deberíamos hablar de *ratón* sino de *dispositivo señalador* o *dispositivo apuntador*. Esto engloba ratón, *touchpad*, pantalla táctil, *trackball*, etc

Algunas definiciones sobre eventos en JavaScript en el DOM

- ¿Qué pasa?
  event type. También llamado *nombre*. Es una cadena de texto que especifica de qué evento se trata. P.e. *click*, *mouseover*, *keydown*, etc
- ¿Dónde pasa?
  event target (objetivo). Elemento que recibe el evento. Todo elemento de una página HTML puede recibirlos: botones, párrafos, enlaces, imágenes, formularios, tablas, etc
- ¿Cómo responder?
  event handler (manejador). Función que se ejecutará cuando se reciba el evento sobre el *target*
- Más detalles
  event object. Objeto con detalles sobre el evento: coordenadas del ratón, tecla pulsada, etc

Cómo registrar manejadores de eventos

En JavaScript contemporáneo, para registrar un manejador de evento

- Seleccionamos el objeto que recibe el evento con el método `document.querySelector()`
  Su argumento será un selector CSS que especifique el elemento
- Invocamos al método `addEventListener` de este objeto, pasando como parámetros
  - El *event type* (nombre del evento)
  - El manejador
- Un mismo evento puede disparar varios manejadores, basta con llamar varias veces a `addEventListener`. Luego se ejecutarán en el orden en que fueron registrados los manejadores

- `querySelector()` devuelve el primer elemento que encaje en el selector (o null si no encaja ninguno)
- `querySelectorAll` devuelve todos los elementos que encajen en el selector

```
<button id="boton01">Dame un clic</button>
<script>
  'use strict'
  function manej_boton01() {
    console.log("Clic recibido.");
  }

  let b = document.querySelector("#boton01");
  // Atención a la almohadilla, es imprescindible
  b.addEventListener("click", manej_boton01);
  // Atajo para ver los log: F12
  // Atajo alternativo: Ctr Shift I
</script>
```

http://ortuno.es/hola_js_01.html

Todo manejador recibe un objeto *event* con los detalles del evento

- Como en el ejemplo anterior, tal vez no lo necesitemos. En ese caso no hace falta declararlo
- En esta versión del ejemplo anterior, sí lo declaramos y lo usamos (para trazarlo)

```
<button id="boton01">Dame un clic</button>
<script>
  'use strict'
  function manej_boton01(event) {
    console.log("Clic recibido.");
    console.log(event);
  }

  let b = document.querySelector("#boton01");
  b.addEventListener("click", manej_boton01);
</script>
```

http://ortuno.es/hola_js_02.html

Depuración

Para depurar un programa JavaScript en el navegador

- Tendremos siempre abierta la consola de logs del navegador, de lo contrario no veríamos ni siquiera los errores que se puedan producir
  - Atajo de teclado en Google Chrome: `Ctr Shift I`
- Escribiremos trazas con `console.log()`
  - Mientras seamos principiantes, es recomendable añadir trazas continuamente, sin esperar a que el programa falle. Cuando el programa funcione aparentemente bien, podremos quitarlas

Un evento que llega a un objeto *element*, se propaga a todos los antecesores de ese elemento. Ejemplo:

1. Un div tiene una tabla que tiene un botón que tiene una imagen
2. El usuario hace clic sobre la imagen
3. El evento llega a la imagen, al botón, a la tabla y al div

Aunque todos los antecesores reciben el evento, sabremos cuál ha sido el *primero* (en este ejemplo, la imagen) por el atributo *event.target*

Técnicas obsoletas

En código, recetas y libros antiguos podremos encontrar selección de elementos con métodos como

```
document.getElementById
document.getElementsByName
document.getElementsByTagName(
document.getElementsByClassName
document.images
document.forms
document.links
... etc
```

Evitaremos estas formas, en favor de `document.querySelector()` y `document.querySelectorAll()`

Tratamiento obsoleto de eventos:

- Métodos *onclick*, *onmouseover*, *onload* ... muchos otros métodos con nombre `on + event type`
- Manejadores de eventos directamente en el HTML

```
<figure class="zoom" onmousemove="zoom(event)"
 style="background-image: url(//blah.com/image/a.jpg)">
```

Evitaremos estas formas, en favor de `addEventListener()`

Texto de un elemento

Para acceder al contenido de un elemento en texto plano, cada objeto *element* tiene la propiedad *textContent*, que podemos leer o escribir

```
<button id="boton01">Registrar hora </button>
<p id="parrafo01">--</p>

<script>
  'use strict'
  function manej_boton01(event) {
    let parrafo01 = document.querySelector("#parrafo01");
    let fecha = new Date();
    parrafo01.textContent = fecha;
  }
  let boton01 = document.querySelector("#boton01");
  boton01.addEventListener("click", manej_boton01);
</script>
```

http://ortuno.es/texto.html

Atributos inexistentes

Recuerda esta característica muy desafortunada de JavaScript

- Si intentamos acceder a una propiedad de un objeto que no exista, el motor no dará ningún error. Simplemente valdrá *undefined*
- P.e. si por error en vez de escribir `p.textContent` escribimos `p.textContext` el motor no nos avisará (al contrario de lo que pasaría en muchos otros lenguajes)

Para el principiante es recomendable poner muchas trazas en la escritura inicial, sin esperar a los errores

- Una traza al comienzo de cada manejador
- Una traza por cada elemento seleccionado
  - Olvidar la almohadilla al seleccionar elementos por su *id* es un error muy frecuente. En este error u otros similares, verás una traza vacía

Naturalmente, trazas tan detalladas acaban siendo molestas

- Bórralas cuando el programa parezca funcionar
- Según vayas adquiriendo experiencia, traza solo aquellos aspectos que te parezca que tengan especial relevancia o dificultad

Todo esto es aplicable a cualquier lenguaje de programación, no solo a JavaScript en el navegador

El ejemplo anterior podrías escribirlo inicialmente así:

```
function manej_boton01(event) {
  console.log("manej_boton01");
  let parrafo01 = document.querySelector("#parrafo01");
  console.log(parrafo01);
  let fecha = new Date();
  console.log("Valor de la fecha:"+fecha);
  parrafo01.textContent = fecha;
}
let boton01 = document.querySelector("#boton01");
console.log(boton01);
boton01.addEventListener("click", manej_boton01);
```

http://ortuno.es/texto.trazas.html

Separación lógica de negocio / interface de usuario

- Es fundamental tener en cuenta que estamos tratando solo con el interface de usuario.
- La lógica de negocio de la aplicación debe estar bien diferenciada, en un código independiente que desarrollaremos y probaremos por separado
  - Excepto tal vez si se trata de funcionalidad trivial

Esto es un ejemplo de lo que **nunca deberíamos hacer**

```
function manej_boton02(event){
    console.log("manej_boton02");
    let div_v_in = document.querySelector("#v_in");
    let v_in = div_v_in.textContent;
    let v_out=v_in*3.6+"Km/h";    //  ¡MUY MAL!

    console.log(v_out);
    let v_out_div = document.querySelector("#v_out");
    v_out_div.textContent = v_out;
}
```

http://ortuno.es/calculo_mal.html

El ejemplo anterior en principio *funciona*, pero

- Mezcla continuamente la programación de botones y displays con los cálculos propios de la aplicación
- Por ser un ejemplo *de juguete* podría pensarse que no tiene importancia
- Pero en un programa real esto acabará dando muchos problemas. Cualquir retoque, corrección o ampliación será mucho más complejo porque afectará a todo el programa en su conjunto

Este tipo de manejadores deben limitarse a

- Leer los valores de entrada desde el HTML
- Llamar al código que realiza los cálculos
- Escribir los valores de salida en el HTML

Enfoque correcto:

```
let v_in = document.querySelector("#v_in").textContent;
let v_out = calcula_velocidad(v_in,"Km/h");

let display = document.querySelector("#v_out");
display.textContent = v_out;
```

Variables globales

Observa el siguiente ejemplo

```
<body>
  <script>
    let x = 0; // Variable global a todo el documento
  </script>
  <div id="display01"></div>
  <br>
  <button id="boton01">Suma 1</button>
  <button id="boton02">Suma 5</button>
  <script>
    'use strict'
    function actualiza_valor() {
      let div = document.querySelector("#display01");
      div.textContent = x;
    }
  </script>
```

```
  <script>
    use 'strict'
    function manej_boton01() {
      x = x + 1;
      actualiza_valor();
    }

    function manej_boton02() {
      x = x + 5;
      actualiza_valor();
    }

    let boton01 = document.querySelector("#boton01");
    boton01.addEventListener("click", manej_boton01);
    let boton02 = document.querySelector("#boton02");
    boton02.addEventListener("click", manej_boton02);
  </script>
</body>
```

http://ortuno.es/globales.html

Aqui la *lógica de negocio* es tan sencilla que, excepcionalmente, la hemos dejado en el manejador

- La comunicación de un valor entre los manejadores la hacemos con una variable global.
  - Error frecuente de principante: declarar la variable dentro de una función. Entonces ya no es global y no funciona nada
- Las variables globales exigen mucha atención, son muy peligrosas. Algunas metodologías las prohíben, el resto, exige minimizarlas
- No deberíamos tener muchas variables globales por separado. Es preferible un único / unos pocos objetos globales, con los atributos necesarios
- Recuerda que estas variable son **única y exclusivamente** para comunicar valores entre manejadores

Creación dinámica de elementos

Con el método *textContent*, hemos visto cómo modificar el texto de un elemento preexistente
Para crear elementos dinámicamente y añadirles cualquier contenido

- `document.createElement()`
  - Recibe como argumento una cadena con el nombre de un tipo de elemento HTML (p, img, table, etc)
  - Devuelve un elemento de ese tipo
- Con el método *append* de un elemento, añadimos otro elemento en su interior

```
<body>
  <button id="boton01">Registrar hora </button>
  <div id="div01">--</div>

  <script>
    'use strict'
    function manej_boton01(event){
        let div01 = document.querySelector("#div01");
        let fecha = new Date();
        let parrafo = document.createElement("p");
        parrafo.textContent = fecha;
        div01.append(parrafo);
    }
    let boton01 = document.querySelector("#boton01");
    boton01.addEventListener("click", manej_boton01);

  </script>
</body>
```

http://ortuno.es/nuevo_p.html

Diferencia textContent append

- textContent
  Es una propiedad
  ```
  parrafo.textContent = fecha;
  ```
  Reemplaza (machaca el valor previo). Solo sirve para texto
- append
  Es un método
  ```
  div01.append(parrafo);
  ```
  Añade cualquier cosa al elemento: texto, una fila, una imagen, un enlace...

En ocasiones puede resultar equivalente usar uno o usar otro. P.e. meter texto en un párrafo vacio o en un texto vacio

```
<button id="boton01">append</button>
<button id="boton02">textContent</button>
<br>
<div id="div01"></>
<script>
  'use strict'
  function usa_append(event){
      let div = document.querySelector("#div01");
      div.append("hola");
  }

  function usa_textContent(event){
      let div = document.querySelector("#div01");
      div.textContent = "hola";
  }

  let boton01 = document.querySelector("#boton01");
  boton01.addEventListener("click", usa_append);

  let boton02 = document.querySelector("#boton02");
  boton02.addEventListener("click", usa_textContent);
</script>
```

http://ortuno.es/append_textContent.html

Insertar una fila en una tabla

Hemos visto cómo insertar un párrafo. De la misma forma, podemos inserta cualquier elemento en cualquier elemento. P.e una fila en una tabla

```
<table id=tabla_horas>
  <tr>
  <th>Hora del clic</th>
  </tr>
</table>
<button id="boton01">Registrar hora </button>
```

```
<script>
  'use strict'
  function manej_boton01(event){
      // Seleccionamos la tabla
      let tabla_horas = document.querySelector("#tabla_horas");

      let tr = document.createElement("tr");  // Creamos la fila

      let td = document.createElement("td");  // Creamos la celda
      let fecha = new Date();
      td.append(fecha); // Metemos la fecha en la celda

      tr.append(td);   // Metemos la celda en la fila
      tabla_horas.append(tr);  // Metemos fila en la tabla
  }
  let boton01 = document.querySelector("#boton01");
  boton01.addEventListener("click", manej_boton01);
</script>
```

http://ortuno.es/nueva_fila.html

O dos celdas en una fila

```
let contador_clics = 0;
function manej_boton01(event){
    let tabla_horas = document.querySelector("#tabla_horas");

    contador_clics = contador_clics + 1;
    let tr = document.createElement("tr"); // Creamos fila

    let td = document.createElement("td"); // Creamos celda
    td.append(contador_clics);  // Metemos contador en celda
    tr.append(td);  // Añadimos celda a fila

    td = document.createElement("td"); // Creamos celda
    let fecha = new Date();
    td.append(fecha);  // Metemos fecha en celda
    tr.append(td);  // Metemos celda en fila

    tabla_horas.append(tr); // Metemos fila en tabla
}
let boton01 = document.querySelector("#boton01");
boton01.addEventListener("click", manej_boton01);
```

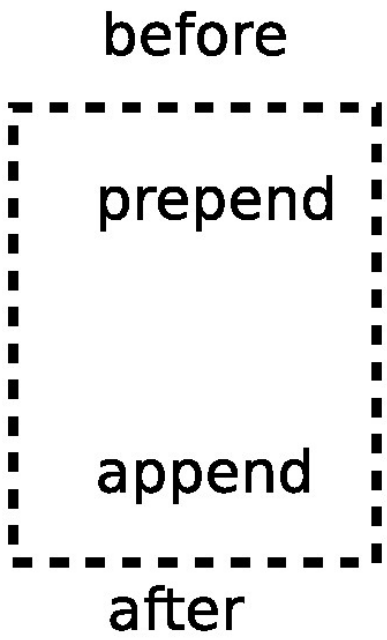
http://ortuno.es/nueva_fila_celdas.html

Añadir contenido

Una vez seleccionado un elemento o elementos mediante un selector, insertarlo en diferentes posiciones

- append()
  Inserta contenido al final de la selección, dentro de la selección
- prepend()
  Inserta contenido al principio de la selección, dentro de la selección
- after()
  Inserta contenido inmediatamente después de la selección, fuera de la selección
- before()
  Inserta contenido inmediatamente antes de la selección, fuera de la selección

En esta figura representamos con una caja punteada el elemento señalado



- *before()* escribe inmediatamente antes
- *prepend()* escribe dentro, al principio
- *append()* escribe dentro, al final
- *after()* escribe inmediatamente después

También cambiarlo o eliminarlo

- replacewith()
  Reemplaza la selección
- remove()
  Borra la selección

Creación de una imagen

Con `createElement` podemos crear cualquier elemento. Y si necesita atributos, también los añadiremos dinámicamente. Por ejemplo en una imagen, los atributos *src*, *alt*
Para construir dinámicamente algo como esto

```
<img src="images/gato.jpg" alt="Gato común europeo de color naranja">
```

por cada atributo en el elemento HTML hay una propiedad en el objeto element correspondiente en JavaScript, que casi siempre tiene el mismo nombre

```
<img src="images/gato.jpg" alt="Gato común europeo de color naranja">
        let img = document.createElement("img");  // Creamos la imagen
        img.src = "images/gato.jpg";
        img.alt = "Gato común europeo de color naranja";
```

Para especificar dónde está el fichero con la imagen, recuerda lo visto en el tema de HTML sobre el fichero especificado en

```
<img src=...>
```

(o cualquier otra referencia a un fichero)

- Un trayecto que no empieza por barra es relativo al directorio actual
  `images/gato.jpg`
- Un trayecto que empieza por barra es absoluto, cuelga del directorio raiz
  `/images/gato.jpg`
- Un trayecto absoluto que incluye el directorio *home* del usuario como una cadena literal, es un error serio
  `/home/alumnos/jperez/images/gato.jpg`

En algunos casos la propiedad en el objeto element no se llama igual que el atributo en el elemento HTML, por ser una palabra reservada en JavaScript

- El atributo *for* del elemento HTML `<label>` se corresponde con la propiedad `htmlFor` del objeto JavaScript
- El atributo *class* de cualquier elemento HTML se corresponde con la propiedad `className` del objeto JavaScript

En HTML las propiedades siempre son de tipo cadena, en JavaScript se convierten a tipo booleano o cadena, si corresponde

```
<button id="boton01">Crear un gato</button>
<div id=div01>
</div>
<script>
  'use strict'
  function crear_gato(event){
      let div01 = document.querySelector("#div01");

      let img = document.createElement("img");  // Creamos la imagen
      img.src = "images/gato.jpg";
      img.alt = "Gato común europeo de color naranja";
      img.width="300";
      div01.append(img); // Metemos la imagen en el div
  }
  let boton01 = document.querySelector("#boton01");
  boton01.addEventListener("click", crear_gato);
</script>
```

http://ortuno.es/crea_imagen.html

Identificación de un elemento seleccionado

La propiedad `event.target` también puede ser útil para saber qué elemento ha recibido un click

- Recuerda que cualquier elemento puede recibir el evento *click*, no hace falta que sea un botón
- Podemos identificar el elemento que recibe el clic con `event.target.id`
- Naturalmente, será necesario que el elemento tenga un atributo *id*, que se lo habremos añadido o bien de forma estática (en el HTML) o bien de forma dinámica (programándolo en JavaScript)

En el siguiente ejemplo, cuando creamos una imagen

- Le añadimos un *id*
- Le añadimos un manejador para que cada vez que la imagen reciba un click, dispare cierta función
- Esa función identifica la imagen a través de `event.target.id`

```
let contador_gatos = 0;
function crear_gato(event){
    let div01 = document.querySelector("#div01");

    let img = document.createElement("img");  // Creamos la imagen
    img.src = "images/gato.jpg";
    img.alt = "Gato común europeo de color naranja";
    img.width="300";

    // Añadimos un id a la imagen
    contador_gatos = contador_gatos + 1;
    let id = "gato_" + String(contador_gatos) ;
    img.id = id;

    // Añadimos un manejador al evento click sobre la imagen
    img.addEventListener("click", identifica_foto)
    div01.append(img); // Metemos la imagen en el div
}
```

Observa que *contador_gatos* tenemos que declararla necesariamente fuera de la función *crear_gato()*

```
function identifica_foto(event){
    let id_foto = event.target.id;
    let span01 = document.querySelector("#span01");
    span01.textContent = id_foto;
}
```

http://ortuno.es/identifica_imagen.html

Modificación de un elemento

Podemos cambiar cualquier elemento dinámicamente, modificando sus atributos

```
<script>
  'use strict'
  pon_gato(); // Empezamos poniendo una imagen para que no quede
              // en la página una foto vacía, incorrecta

  function pon_gato(event){
      let img = document.querySelector("#foto");
      img.src = "images/gato.jpg";
      img.alt = "Gato común europeo de color naranja";
      img.width="300";
  }

  function pon_periquito(event){
      let img = document.querySelector("#foto");
      img.src = "images/periquito.jpg";
      img.alt = "Periquito azul";
      img.width="300";
  }
</script>
```

```
<script>
  let boton01 = document.querySelector("#boton01");
  boton01.addEventListener("click", pon_gato);

  let boton02 = document.querySelector("#boton02");
  boton02.addEventListener("click", pon_periquito);
</script>
```

http://ortuno.es/cambia_imagen.html

Es frecuente que queramos cambiar dinámicamente el aspecto de un documento HTML, esto es, sus atributos CSS. P.e. quitar o poner el valor *none* al atributo *display*
Aunque se puede modificar directamente un atributo CSS, normalmente es preferible

- Crear una regla CSS que modifique el atributo para cierta clase
- Quitar y poner la clase [11]

```
<style>
  .oculto {
    display: none;
  }
</style>
```

[11] De la misma forma que en un procesador de texto podemos modificar el formato de un párrafo directamente, pero en general es preferible asignar un estilo al párrafo, y modificar el formato del estilo

```
<button id="boton01">Ver foto</button>
<button id="boton02">Quitar foto</button>
<div id="marco_foto" class="oculto">
  <img src="images/plaza_espana.jpg" alt="Plaza de España, Madrid">
</div>
<script>
  'use strict'
  function quita_clase() {
      console.log("quita_clase");
      let marco = document.querySelector('#marco_foto');
      marco.classList.remove("oculto");
  };

  function pon_clase() {
      console.log("pon_clase");
      let marco = document.querySelector('#marco_foto');
      marco.classList.add("oculto");
  };

  let boton01 = document.querySelector("#boton01");
  boton01.addEventListener("click", quita_clase);
  let boton02 = document.querySelector("#boton02");
  boton02.addEventListener("click", pon_clase);
</script>
```

En el ejemplo anterior, lo que buscamos es alternar entre esto

```
<div id="marco_foto" class="oculto">
  <img src="images/plaza_espana.jpg" alt="Plaza de España, Madrid">
</div>
```

y esto

```
<div id="marco_foto">
  <img src="images/plaza_espana.jpg" alt="Plaza de España, Madrid">
</div>
```

- Esto es, añadir y quitar la clase *oculto* al div cuyo identificador es *marco_foto*
- Esto es, añadir y quitar el atributo `class` con el valor `oculto`

Desde el punto de vista sintático, *class* es un atributo HTML como otro cualquiera, pero tiene algunas características que suelen despistar al principiante

- Posiblemente el nombre *class* es poco afortunado. Sería más coherente *classes*, porque es una lista de elementos
- Como hemos visto, el atributo `class` se corresponde con la propiedad JavaScript `className`, por ser *class* una palabra reservada
- Podemos quitar y poner clases manejando directamente la cadena almacenada en `className`, pero resulta más conveniente el uso de la propiedad `classList`

En el DOM, `classList` es una propiedad de los objetos elemento que devuelve una lista de las clases del elemento

- Esto es, de los valores del atributo *class*, troceados por espacios

En esta lista podemos usar los métodos `add()`, `remove()`, `contains()` y `toggle()`
De esta forma,

- Con el código
  ```
  marco.classList.add("oculto");
  ```
  conseguimos algo equivalente a

  ```
  <div id="marco_foto" class="oculto">
  ```

- Con el código
  ```
  marco.classList.remove("oculto");
  ```
  conseguimos algo equivalente a

  ```
  <div id="marco_foto">
  ```

Con el método `toggle` (alternar) de la lista de atributos

- Si el atributo está incluido en la lista, lo borramos
- Si no está incluido, lo añadimos

```
function alterna_clase() {
    let marco = document.querySelector('#marco_foto');
    marco.classList.toggle("oculto");
};

let boton01 = document.querySelector("#boton01");
boton01.addEventListener("click", alterna_clase);
```

http://ortuno.es/quita_pon_02.html

querySelectorAll

En todos los ejemplos anteriores, seleccionábamos un elemento del DOM (y solo uno) por su identificador. Pero habrá ocasiones en las que necesitamos seleccionar una serie de elementos.
Para ello disponemos del método `document.querySelectorAll`

- No es exactamente un array, sino un objeto similar, *NodeList*
- Si esta lista está vacía, su atributo *length* valdrá 0

En este ejemplo, seleccionamos todos elementos que contengan la clase *gato*, para eliminarlos

```
function borra_gatos(event){
    let gatos = document.querySelectorAll(".gato");
    for (let gato of gatos){
        gato.remove();
    }
}
```

http://ortuno.es/elimina.html

Hay muchas formas de seleccionar una serie de elementos:

- Todos los que tengan cierto atributo, con cierto valor, p.e. el fichero de una imagen
- Todos los que tengan su nombre en una lista
- Todos los que tengan un identificador siguiendo cierto patrón
- ...

Pero en general lo recomendable será seleccionar un elemento por una clase

- Para ello, naturalmente, a la hora de crearlo, nos habremos encargado de añadirle la clase

display / visibility

Para hacer un elemento invisible, puedo usar

- `display`
  Oculta el elemento y reposiciona el resto
- `visibility`
  Oculta el elemento, sin reposicionar el resto

```
<style>
  .sin_display {
    display: none;
  }
  .invisible {
    visibility: hidden;
  }
</style>
...
<script>
  for (let gato of gatos){
      if (usa_display)
          gato.classList.toggle("sin_display");
      else
          gato.classList.toggle("invisible");
      }
</script>
```

http://ortuno.es/selecciona.html

Modificar el CSS directamente

Para modificar los atributos css, normalmente es preferible vincularlos a una clase y poner y quitar clases, como acabamos de ver

- Pero en ocasiones puede ser conveniente modificar los atributos directamente. Cada elemento del DOM tiene una propiedad *style* que a su vez contiene un objeto cuyos atributos son los atributos CSS.
- Con una modificación: los atributos CSS suelen incluir guiones, que no son válidos en JavaScript (se interpretarían como el operador de resta). Así que hay que convertir a notaciónDromedario
  P.e. `border-width` sería `style.borderWidth`

```
function manej_boton01(event) {
  let p01 = document.querySelector("#span01");
  p01.style.backgroundColor="LightSkyBlue";
}

function manej_boton02(event) {
  let p01 = document.querySelector("#span01");
  p01.style.backgroundColor="LightSalmon";
}
```

http://ortuno.es/css_js.html

Eventos de ratón

- *mouseover*
  Se recibe cuando el ratón se coloca sobre el elemento
- *mouseout*
  Se recibe cuando el ratón abandona el elemento o cualquiera de sus descendentes
- *mouseleave*
  Se recibe cuando el ratón abandona el elemento seleccionado

Eventos mouseout, mouseleave

Podemos observar la diferencia entre *mouseout* y *mouseleave* en esta demo. No usa JavaScript contemporáneo sino jQuery, pero podemos seguir el funcionamiento general

`https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_event_mouseleave_mouseout`

En este ejemplo tenemos:

- Un primer *div* de clase *over* que contiene un texto que contiene un *span*
  - A este div se le pone un manejador para el evento *mouseout*
  - Cada vez que el *ratón* abandona el div o que el ratón abandona uno de sus descendientes (el texto), se dispara el manejador y se incrementa el contador
- Un segundo *div* de clase *enter* que contiene un texto que contiene un *span*
  - A este div se le pone un manejador para el evento *mouseleave*
  - Cada vez que el ratón abandona el div, se dispara el manejador. Pero cuando abandona sus descendientes, no

Eventos mouseout, mouseleave (ii)

Observaciones adicionales:
El autor del ejemplo de *w3schools* tiene un estilo de programación diferente al que empleamos en la asignatura. Debemos entenderlo aunque no lo usemos

- Al primer div le pone la clase *over*, al segundo, la clase *enter*
- Para capturar el primer div emplea el selector `div.over`, para el segundo, `div.enter`. Esto es: *el (los) div de clase over, el (los) div de clase enter*
- Para capturar los span emplea los selectores `.over span` y `.enter span`. Esto es: el (los) span que están dentro de un elemento de clase over y el (los) span que están dentro de un elemento de clase enter

En nuestos ejemplos, habríamos usado un id para cada div y para cada span

El ejemplo de *w3schools* también emplea otra técnica que aquí desaconsejamos: usar un valor almacenado en el interface de usuario

- Para incrementar los contadores, captura el valor que hay dentro del *span* y le suma 1. Lo hace en jQuery, en JavaScript contemporáneo usaríamos *textContent*
- El diseño que preferimos aquí emplearía una variable distinta para estos contadores. El span se limitaría a copiar esa variable

En este ejemplo añadimos manejadores de eventos de ratón a un elemento ya creado. Exactamente igual que en todos los ejemplos anteriores

```
function manej01(event) {
  event.target.classList.add("destacado");
}

function manej02(event) {
  event.target.classList.remove("destacado");
}

let p01 = document.querySelector("#p01");
p01.addEventListener("mouseover", manej01);
p01.addEventListener("mouseout", manej02);
```

http://ortuno.es/eventos_01.html

También podemos añadir los manejadores al crear elementos dinámicamente

```
function crea_parrafos(){
  for(let i=0; i<3; ++i){
    let div01 = document.querySelector("#div01");
    let p = document.createElement("p");
    p.textContent = "Lorem ipsum dolor sit amet";
    p.addEventListener("mouseover", manej01);
    p.addEventListener("mouseout", manej02);
    div01.append(p);
  }
}
```

http://ortuno.es/eventos_02.html

Pero si necesitamos un mismo manejador sobre una serie de elementos, no es necesario añadirlo uno a uno a todos ellos

- Si los elementos los creámos dinámicamente esto no debería ser problema, pero si son elementos preexistente, puede resultar un poco pesado

Recuerda que un evento que llega a un elemento, también llega a todos sus antecesores. Podemos:

- Poner en el manejador en un antecesor común (llamémosle *abuelo*)
- Identificar en el *abuelo* al *nieto* concreto donde se disparó el evento, con *event.target*. De esta forma, el manejador está en el *abuelo* pero la acción se realiza en el *nieto*

```
<div id=div01>
  <p id=p01>Lorem ipsum dolor sit amet.</p>
  <p id=p02>Lorem ipsum dolor sit amet.</p>
  <p id=p03>Lorem ipsum dolor sit amet.</p>
</div>
<script>
  function manej01(event) {
    event.target.classList.add("destacado");
  }

  function manej02(event) {
    event.target.classList.remove("destacado");
  }

  let div = document.querySelector("#div01");
  div.addEventListener("mouseover", manej01);
  div.addEventListener("mouseout", manej02);
</script>
```

http://ortuno.es/eventos_03.html

Formularios (1)

Para acceder al valor de un formulario

- Añadimos un manejador para el evento *change*
- Leemos el atributo *value* del objeto *element* correspondiente

```
<form action="/action_page.html" id=formulario01>
  <label for="usuario">Usuario</label>
  <input type="text" name="usuario" id="usuario">
</form>
<script>
  function manej01(event) {
    let usuario = document.querySelector("#usuario");
    console.log(usuario.value);
  };
  let formulario01 = document.querySelector("#formulario01");
  formulario01.addEventListener("change", manej01);
</script>
```

http://ortuno.es/formulario_01.html

Formularios (2)

Para consultar el estado de un *radiobutton* o un *checkbox*, usamos un selector como

`input[name=figura]:checked`

Captura todos los *input* que contengan el atributo *name* con el valor *figura*
El selector `:checked` captura todos los `<input>` activados

```
  <form id=formulario01 action="/action_page.html">
    <input type="radio" name="figura" value="sota">Sota
    <input type="radio" name="figura" value="caballo">Caballo
    <input type="radio" name="figura" value="rey">Rey
  </form>
  <script>
    'use strict'
    function manej01(event) {
      let input =
↪   document.querySelector("input[name=figura]:checked");
      console.log(input.value);
    };
    let formulario01 = document.querySelector("#formulario01");
    formulario01.addEventListener("change", manej01);
  </script>
```

http://ortuno.es/formulario_02.html

Validación de un formulario

Validar un formulario consiste en comprobar que el usuario ha rellenado los campos con valores adecuados. Esto podemos hacerlo

- Con métodos específicos de HTML
- Con código JavaScript, como el siguiente ejemplo:
  Cuando el usuario acabe de escribir el valor para la contraseña (cuando cambie el foco a otro *input* o pulse *enviar*), se disparará el evento *change* que disparará el manejador para comprobar que la contraseña tenga la longitud mínima exigida

```
let contrasenia_minima = 8;

function manej01(event) {
  let display = document.querySelector("#validacion");
  let contrasenia = document.querySelector("#contrasenia").value;
  if (contrasenia.length >= contrasenia_minima) {
    display.textContent = "Contraseña aceptable";
  } else {
    display.textContent = "Contraseña muy corta";
  }
};
let campo_contra = document.querySelector("#contrasenia");
campo_contra.addEventListener("change", manej01);
```

http://ortuno.es/validacion_01.html

Podemos mejorar el ejemplo anterior, de forma que no sea necesario que el usuario acabe de editar un campo para obtener realimentación. Para ello capturamos los eventos

- `Change`
  Fin de edición del *input*
- `keyup`
  Pulsación de teclado (liberación de la tecla pulsada)
- `paste`
  Pegado desde el portapapeles
- `mouseup`
  Pulsación del botón del ratón (fin de la pulsación)

```
function manej01(event) {
  let contrasenia = document.querySelector("#contrasenia").value;
  valida_contrasenia(contrasenia);
};
let campo_contra = document.querySelector("#contrasenia");
campo_contra.addEventListener("change", manej01);
campo_contra.addEventListener("keyup", manej01);
campo_contra.addEventListener("paste", manej01);
campo_contra.addEventListener("mouseup", manej01);
```

http://ortuno.es/validacion_02.html

JavaScript (ii)

Números aleatorios

```
'use strict'
// Math.floor() trunca un número real
console.log(Math.floor(9.99));  // 9

// Math.round () redondea un número
console.log(Math.round(9.5));  // 10
console.log(Math.round(-9.4));  // -9
console.log(Math.round(-9.5));  // -9
console.log(Math.round(-9.6));  // -10

// Math.random() devuelve un número real
// aleatorio entre 0 (inc) y 1 (excl)
console.log(Math.random());

let k = 10;
let x;

//Número entero aleatorio entre 0(inc) y k (excl)
x = Math.floor(k * Math.random());
console.log(x);
```

http://ortuno.es/ej_random.js

Fecha y Hora

La fecha (con su hora) se guarda en objetos de tipo `Date`

- Siempre es una hora UTC (Coordinated Universal Time), anteriormente llamada hora de Greenwich
- Se guarda como milisegundos transcurridos desde el *epoch*, 1 de enero de 1970
  - Es recomendable almacenar, procesar y transmitir este formato en todo momento, y solo inmediatamente antes de presentarlo al usuario, realizar la conversión necesaria
- Atención, no es el *tiempo unix* (segundos transcurridos desde el *epoch*), como en otros sistemas. Aquí son milisegundos
- Hay muchas páginas web que permiten conocer y manipular el tiempo unix. Busca en internet *unix time converter*

```
'use strict'

// Si construimos un objeto Date sin pasar argumentos,
// obtenemos la fecha actual
let d1 = new Date();
console.log(d1); // 2020-04-22T10:24:16.528Z

// Hacemos un cálculo matemático cualquiera, complejo,
// para tardar cierto tiempo (en js no hay nada esquivalente
// a sleep)

for (let i = 0; i<100000000; ++i){
    let x = Math.random();
    x**2.50;
}

let d2 = new Date();
console.log(d2);  // 2020-04-22T10:24:17.555Z

// Tiempo transcurrido
console.log(d2-d1);  // 1027  (milisegundos)
```

Con frecuencia construiremos los objetos Date a partir de *intervalos*

- Un intervalo es un objeto de tipo *number* que contiene los milisegundos transcurridos desde el 1 de enero de 1970
- Conceptualmente es igual que un Date, pero es un tipo distinto (tipo *number*, no tipo *date*)
- Recuerda: muy parecido a una hora Unix, pero expresado en milisegundos

```
'use strict'
let intervalo = 0;
let d1 = new Date(intervalo);

console.log(intervalo, typeof(intervalo));
    // 0 'number'

console.log(d1, typeof(d1));
    // 1970-01-01T00:00:00.000Z 'object'
```

```
'use strict'

// Los intervalos son prácticos para hacer cálculos
let intervalo1 = 0;
let intervalo2 = intervalo1 + 24 * 60 * 60 * 1000;
     // ms en un día

console.log(intervalo1);   // 0
console.log(intervalo2);   // 86400000
let d1 = new Date(intervalo1);
console.log(d1);           // 1970-01-01T00:00:00.000Z
let d2 = new Date(intervalo2);
console.log(d2);          // 1970-01-02T00:00:00.000Z
```

Si necesitamos construir un *date* a partir del año, mes, día, etc, podemos pasar estos argumentos al constructor

- Pero cuidado, estos parámetros deberán ser hora local
- Si necesitamos una hora concreta a partir de la hora UTC, es necesario construir un intervalo con `Date.UTC()` y luego usarlo para construir *date*

```
// El constructor 'Date' devuelve un objeto Date, a partir de
// la hora local. P.e. 3 de septiembre de 2018, a las 9 de
// España. Cuidado: 0 es enero, 11 es diciembre
d = new Date(2018, 8, 3, 9, 0, 0);
console.log(d); // 2018-09-03T07:00:00.000Z   ZULU time (UTC)

// Para crear un Date desde hora UTC, usamos la función Date.UTC
// Atención: no devuelve un Date, devuelve un intervalo que
// debemos usar para crear el Date
let intervalo3 = Date.UTC(2018, 8, 3, 9, 0, 0);
console.log(intervalo3);  //  1535965200000

// Ahora construyo el objeto Date desde el intervalo
let d3 = new Date(intervalo3)
console.log(d3);  // 2018-09-03T09:00:00.000Z

// No se puede construir el objeto directamente
// d3 = new Date.UTC(2018, 8, 3, 9, 0, 0);
// TypeError: Date.UTC is not a constructor
```

```
d=new Date(2017, 8, 1, 9, 0, 0);  // 9:00 hora española
                                  // Mes 8: septiembre

// Acceso a cada unidad, expresada como hora local
console.log(d.getFullYear());   // 2017
console.log(d.getMonth());  // 8 (septiembre)
console.log(d.getDate());  // 1
console.log(d.getDay()); //5  (viernes)
console.log(d.getHours()); // 9  hora española
console.log(d.getMinutes()); // 0
console.log(d.getSeconds()); // 0
console.log(d.getMilliseconds()); //

// Acceso a cada unidad, expresada como hora UTC
console.log(d.getUTCFullYear());   // 2017
console.log(d.getUTCMonth());  // 8 (septiembre)
console.log(d.getUTCDate());  // 1
console.log(d.getUTCDay()); //5  (viernes)
console.log(d.getUTCHours()); // 7  hora UTC
console.log(d.getUTCMinutes()); // 0
console.log(d.getUTCSeconds()); // 0
console.log(d.getUTCMilliseconds()); // 0
```

Cálculo del tiempo transcurrido entre dos fechas

```
'use strict'
let d1,d2;

// 1 de septiembre de 2017, a las 9 de España
d1=new Date(2017, 8, 1, 9, 0, 0);

// las 9 y 10
d2=new Date(2017, 8, 1, 9, 10, 0);

console.log(d2-d1) // 600000 (600 segundos)
```

```
'use strict'
let d1,d2,s, ms_año, edad;

//  8 de julio de 1996, creación de la URJC
d1=new Date(1996, 6, 8, 0, 0, 0);
d2=new Date();
console.log(d1); //  1996-07-07T22:00:00.000Z
console.log(d2); // 2020-04-20T15:04:35.097Z

s= (d2-d1) ;  // milisegundos transcurridos

ms_año= 31536000000  // 1000*60*60*24*365
edad= (s/ms_año).toFixed(2);  // Redondeo a 2 decimales

console.log('Edad de la URJC:', edad);   // 23.80
```

Hora Unix a partir de Date

```
'use strict'

let d1 = new Date();
console.log(d1);   // 2020-04-22T13:38:34.777Z

// Basta con restar 0 y convertir a segundos
console.log( (d1-0) / 1000);       // 1587562714.777

// O usar el método getTime(), que es equivalente
console.log(d1.getTime() / 1000);  // 1587562714.777
```

Excepciones

Las excepciones son similares a las de cualquier otro lenguaje

- Con la particularidad de que el argumento de `throw` (la excepción), puede ser una cadena, un número, un booleano o un objeto

```
'use strict'
// Atención, ejemplo no realista

try {
  throw 'xxx27';
} catch (e) {
  console.log('capturada excepción ' + e);
     // capturada excepción xxx27
}
```

Cuando el motor de JavaScript lanza una excepción, es un objeto, con las propiedades `name` y `message`

```
'use strict'
// Atención, ejemplo no realista

try{
    console.log( no_definido);
} catch (e) {
    console.log("capturada excepción ",e.name,e.message);
}
//capturada excepción  ReferenceError no_definido is not defined
```

Captura de un tipo concreto de excepción. P.e. `ReferenceError`

```
'use strict'
// Atención, ejemplo no realista
try {
  console.log(no_definido);
} catch (e) {
  console.log("Capturada excepción");
  switch (e.name) {
    case ('ReferenceError'):
      console.log(e.name + " Objeto no definido");
      break;
    default:
      console.log(e.name + " Excepción inesperada");
      break;
  }
}
```

Hay 7 nombres de error

- Error
  Unspecified Error
- EvalError
  An error has occurred in the eval() function
- RangeError
  A number *out of range* has occurred
- ReferenceError
  An illegal reference has occurred
- SyntaxError
  A syntax error has occurred
- TypeError
  A type error has occurred
- URIError
  An error in encodeURI() has occurred

Aunque nuestros programas pueden lanzar simplemente cadenas, números o booleanos, es preferible lanzar objetos

Hay un constructor para cada nombre de error

```
'use strict'
// Atención, ejemplo no realista

try{
    throw new RangeError('Problema en módulo xxx28');
} catch (e) {
    console.log('capturada excepción ',e.name,e.message);
    // capturada excepción  RangeError Problema en módulo xxx28
}
```

¡Muy importante!

- El código de ejemplo que hemos visto, donde lanzamos la excepción e inmediatamente la capturamos, no es realista. Solo sirve para ilustrar el funcionamiento de las excepciones
- En situaciones reales, no tiene ningún sentido que la misma función lance una excepción y luego la capture.
- Lo normal es que un programa lance la excepción y otro programa distinto la capture. O que no programemos explícitamente la captura de la excepción, con lo que la acaba capturando la shell
- Si la especificación nos pide *lanzar* una excepción, debemos hacer exactamente eso, lanzarla. No lanzarla y luego capturarla

Otro aspecto no realista:

- En los ejemplos previos, capturamos la excepción, mostramos una traza *y nada más*. El programa sigue su curso, a pesar del problema
- Es mucho más frecuente notificar el problema y detener la ejecución del programa, si no es posible corregirlo

Módulos

Un módulo es

- Un fichero que contiene código JavaScript
- Que es invocado desde otro fichero distinto, que contiene el código principal

En otros lenguajes se les llama bibliotecas o librerías

En ECMAScript 5 y anteriores no había una forma nativa de usar módulos. Se desarrollaron diferentes herramientas, incompatibles entre sí, que permitían su uso
Las principales son:

- CommonJS
  Para Node.js
- RequireJS
  Para el navegador

En ECMASCript 6 sí hay soporte nativo para módulos, una mezcla de las dos sintaxis

- En la actualidad, año 2019, tanto en los navegadores como en node.js podemos usar módulos. Pero si el motor es un poco antiguo, esta funcionalidad de ECMAScript 6 puede no estar implementada

Extensiones de los ficheros

El mismo código ECMASCript 6 con módulos podemos ejecutarlo tanto en node.js como en el navegador. Con una diferencia:

- En el navegador, tanto los módulos como los ficheros que usan los módulos, tienen que tener extensión `.js`
- En node.js, tanto los módulos como los ficheros que usan los módulos, tienen que tener extensión `.mjs`

Una solución es usar los enlaces simbólicos de Unix (Linux, macOS). O los accesos directos de Windows

1. Creamos los ficheros con extensión `.mjs` para node.js
2. Para el navegador, creamos enlaces simbólicos `.js`

```
ln -s modulo.mjs modulo.js
ln -s programa.mjs programa.js
```

Si lo hacemos al revés (el fichero original con `.js` y enlace con `.mjs`), no funciona. Node.js toma el `js`, dando error

modulo.mjs

```
// Anteponemos 'export' a las funciones a exportar
export function f(){
    return "efe";
}

export function g(){
    return "ge";
}

// Esta función no la exportamos
function h(){
    return "hache";
}
```

programa.mjs

```
'use strict'
import * as modulo from './modulo';
console.log(modulo.f());
console.log(modulo.g());
```

Los módulos también se pueden usar de esta forma

```
'use strict'
import { f, g } from './modulo';
console.log(f());
console.log(g());
```

Pero aquí lo desaconsejamos, porque la llamada a una función no indica explícitamente de qué módulo viene

Observaciones

- El código dentro de un módulo siempre está en modo estricto, no es necesario indicarlo explícitamente
- Para especificar la ubicación del módulo, es imprescindible indicar el path relativo con `./`
- Si queremos que el código funcione tanto en el navegador como en node.js, no ponemos extensión

```
import * as modulo from './modulo';
```

```
import { f, g } from './modulo';
```

- Si sólo va a funcionar...
  - en node.js, podemos poner extensión `.mjs`

```
import * as modulo from './modulo.mjs';
```

```
import { f, g } from './modulo.mjs';
```

  - en el navegador, podemos poner extensión `.js`

```
import * as modulo from './modulo.js';
```

```
import { f, g } from './modulo.js';
```

Esto será necesario para probar nuestras prácticas de html canvas desde un servidor web local

Uso en node.js

- En las versiones actuales de node.js, el soporte para los módulos es experimental, hay que añadir un flag para interpretarlo

```
node --experimental-modules programa.mjs
```

Uso en el navegador

```html
<!DOCTYPE html>
<html lang="es-ES">

<head>
  <meta charset="utf-8">
  <title>Probando modulos</title>
</head>

<body>
  <script type="module" src="programa.js"> </script>
</body>

</html>
```

http://ortuno.es/probando_modulos.html

- En el elemento *script* añadimos el atributo *type=module*
  No porque `programa.js` sea un módulo, sino porque usa módulos

Un script que usa módulos, integrado en una página web solo puede ejecutarse cuando la página ha sido cargada desde un servidor web, no cuando ha sido leída de un fichero local

- Son restricciones de seguridad de CORS (*Cross-origin resource sharing*)

Para probar tus prácticas, tienes varias opciones
Opción 1: Web Server for Chrome

- Googlea *web server for Chrome*. Es una app para el navegador Chrome. Instálala
- Indícale el directorio a servir: el directorio donde están tus ficheros html
- Con esto tendrás disponibles tus ficheros en un servidor web local, accesible por omisión en `http://127.0.0.1:8887`

Opción 2: Servidor web local

P.e. usando python

`python -m SimpleHTTPServer <PUERTO> <DIRECTORIO>`

Los ficheros estarán accesibles en `http://localhost:<PUERTO>`

- En este caso, tendrás que importar módulos especificando la extensión `.js`, con lo que el código ya no funcionará en node.js

  Ejemplo:

  ```
  import * as vjcanvas from "./vjcanvas.js"
  ```

  en vez de

  ```
  import * as vjcanvas from "./vjcanvas"
  ```

  (Un servidor *de verdad* como p.e. apache o Nginx permite omitir la extensión. Pero el servidor web local necesita extensión explícita)

- `SimpleHTTPServer` es un módulo estándar de python, basta con tener python instalado

TypeScript

Como hemos visto, JavaScript tiene

- Características muy interesantes
- Muchos inconvenientes.
  Forman parte de la esencia del lenguaje, para evitarlos habría que usar un lenguaje diferente

Lenguajes alternativos a JavaScript hay muchos. Uno de los más convenientes es TypeScript

- TypeScript es un lenguaje *Open Source*, de alto nivel. Aparece en el año 2012, desarrollado y mantenido por Microsoft. Diseñado para hacer aplicaciones Web, en el cliente y en el servidor
- Es un superconjunto de JavaScript, al que añade tipado estático
  - En otras palabras: JavaScript es un subconjunto de TypeScript. Todo programa en JavaScript es también un programa en TypeScript

- El código fuente en TypeScript se *transpila* a JavaScript, de la misma forma que el código fuente de otros lenguajes se compila en código objeto / ejecutable. Por tanto, TypeScript funciona sobre cualquier navegador
- TypeScript permite detectar la práctica totalidad de problemas que *se le escapan* a JavaScript

POO basada en herencia vs POO basada en prototipos

La gran mayoría de lenguajes y herramientas diversas que emplean POO (programación orientada a objetos) aplican POO basada en herencia

- Es la forma tradicional. De hecho, es frecuente considerar que *POO* necesariamente implica herencia

En POO tradicional, hay un principio generalmente aceptado:
Favor object composition over class inheritance
(E. Gamma et al. *Design Patterns*. 1994)
La POO basada en prototipos va un paso más allá

- Para solucionar una serie de problemas bien conocidos en la POO tradicional. No *prefiere* la composición frente a la herencia, sino que omite por completo la herencia y se basa solo en composición.

Problemas de la herencia (1)

- Código yo-yo
  En jerarquías de cierta longitud, el programador se ve obligado a subir y bajar por las clases continuamente
- La herencia es una relación fuertemente acoplada. Los hijos heredan todo sobre sus padres, necesitan conocer todo sobre sus padres (y abuelos, bisabuelos...)
- Problema del gorila y el plátano
  Yo solo quería un plátano, pero me dieron el plátano, el gorila que sostenía el plátano y la jungla entera

Problemas de la herencia (2)

- La herencia es inflexible
  Por muy bien que se diseñe una jerarquía de clases, en dominios medianamente complejos acaban apareciendo casos no previstos que no encajan en la taxonomía inicial
- Herencia múltiple
  Heredar de diferentes clases es deseable, y teóricamente posible. Pero en la práctica resulta muy complicado
- Arquitectura frágil
  Rediseñar una clase obliga a modificar todos sus descendientes

POO basada en prototipos

- Aparece en el lenguaje Self, a mediados de los años 1980
- No hay clases como instancias de objetos: solo hay objetos, que se producen mediante funciones denominadas *factorías de objetos*
- Un objeto es una unidad de código que soluciona un problema concreto. No es necesario (o al menos no es crítico) pensar cómo usar variantes de ese objeto en otros casos
- A partir de ese objeto, según se va necesitando, se crean nuevos objetos añadiendo o eliminando las propiedades (datos y métodos) necesarias.
- Metáfora.
  Herencia: piezas de Ikea
  Prototipo: piezas de Lego

Problemas de la POO basada en prototipos (1)

- Poco conocida
  Pocos programadores la usan bien, pocos libros la explican bien
- No adecuada para principiantes
  Una vez que se conoce su uso resulta sencillo, pero su curva de aprendizaje es pronunciada
  Ejemplo: en JavaScript es necesario manejar al menos los siguiente conceptos:
  - Lambdas
  - Cierres
  - Funciones flecha
  - Prototipos
  - Factoria de objetos
  - Extensiones dinámicas de objetos: mezclas, composición, agregación

Problemas de la POO basada en prototipos (2)

- La flexibilidad del paradigma añade complejidad al intérprete/compilador, lo que afecta al rendimiento
  - Aunque los motores modernos son muy eficientes y este problema solo es relevante en casos extremos
- La flexibilidad del paradigma puede hacer más difícil el garantizar que los resultados sean correctos
  - Crítica análoga a la que hacen los partidarios de lenguajes de tipado estático frente a lenguajes de tipado dinámico
  - Una clase es un contrato estricto sobre lo que puede o no puede hacer un objeto. Estas restricciones no existen con los prototipos
    With great power comes great responsibility

Enlaces sobre POO basada en prototipos

- Composition over Inheritance
  Video en YouTube del canal *Fun Fun Funtions*
  `https://youtu.be/wfMtDGfHWpA`
- Master the JavaScript Interview: What's the Difference Between Class & Prototypal Inheritance?
  Eric Elliott
  `http://tinyurl.com/zbtjruf`
- Programming JavaScript Applications
  Eric Elliott. O'Reilly, 2015
  `http://proquest.safaribooksonline.com/book/programming/javascript/9781491950289`
- The Principles of Object-Oriented JavaScript
  Nicholas Zakas. No Starch Press, 2014
  `http://proquest.safaribooksonline.com/book/programming/javascript/9781457185304`

POO basada en herencia en JavaScript

El lenguaje JavaScript usa POO basada en prototipos.
Tras una cierta polémica, ECMASCript 6 soporta POO basada en herencia

- En realidad es *azúcar sintáctico*. Internamente siguen siendo prototipos

Los argumentos principales por los que se acepta este *paso atrás* son

1. Muchos programadores insisten en usarla. Distintas librerías, distintas soluciones ad-hoc. Es preferible una implementación oficial
2. Para principiantes con problemas sencillos, resulta más adecuado

Por este último motivo, en la presente asignatura veremos POO basada en herencia, no POO basada en prototipos

Clases

- Definimos clases con la palabra reservada `class`, el nombre de la clase y entre llaves, sus propiedades
- Por convenio, los nombres de clase empiezan por letra mayúscula
- A diferencia de lo que ocurre en los objetos, las propiedades no van separadas por comas
- Las propiedades de los objetos se crean en el método `constructor()`
- Se accede a las propiedes mediante la palabra reservada `this`, que representa al objeto
- Para crear una clase heredera de otra:

```
class Hija extends Madre{ ....}
```

- Para llamar a un método de la clase padre, se usa la palabra reservada `super`

```
'use strict'
class Circunferencia{
    constructor(x,y,r){
        this.x=x;
        this.y=y;
        this.r=r;
    }
    aCadena(){
        return '('+this.x+','+this.y+','+this.r+')';
    }
}
class Circulo extends Circunferencia{
    constructor(x,y,r,color){
        super(x,y,r);
        this.color=color;
    }
    aCadena(){
        return super.aCadena()+ " color:"+ this.color;
    }
}
let a=new Circunferencia(2,2,1);
console.log(a.aCadena());   // (2,2,1)
let b=new Circulo(2,2,1,"azul");
console.log(b.aCadena());  // (2,2,1) color: azul
```

Métodos

Para declarar los métodos de una clase no es necesario usar la palabra reservada `function`
Las clases tienen tres tipos de métodos.

- `constructor()`
  Es un método especial para crear inicializar los objetos de esta clase
- Métodos de prototipo (métodos *normales*)
- Métodos estático. (métodos *de clase*)
  Se crean anteponiendo la palabra reservada `static`
  Se invocan sin instanciar las clases en objetos, no pueden llamarse a través de una instancia de clase (a través de un objeto), solo a través de la clase
  ¿En qué casos un método debería ser estático?
  - Cuando tenga sentido sin que se haya declarado un objeto
  - Cuando procese 2 o más objetos, sin que uno tenga más relevancia que otro

```
'use strict'
class Circunferencia{
    constructor(x,y,r){
        this.x=x;
        this.y=x;
        this.r=r;
    }
    aCadena(){
        return '('+this.x+','+this.y+','+this.r+')';
    }
    static distanciaCentros( a, b){
        return Math.sqrt(
            Math.pow(a.x-b.x, 2) + Math.pow(a.y-b.y,
            2));
    }
    longitud(){
        return 2*Math.PI*this.r;
    }
}

let p=new Circunferencia(0,0,1);
let q=new Circunferencia(1,1,1);
console.log(p.aCadena());  // (0,0,1)
console.log( Circunferencia.distanciaCentros(p,q));
    // 1.4142135623730951
console.log(p.longitud()); // 6.283185307179586
```

Enlaces sobre clases en JavaScript

- MDN Web Docs. Classes

  `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`

- *Exploring ES6. Upgrade to the next version of JavaScript*
  Axel Rauschmayer

  `http://exploringjs.com/es6/ch_classes.html#sec_overview-classes`

APIs de HTML5

HTML 5 define algunas nuevas APIS. Podemos englobar en este conjunto otras tecnologías que aunque en rigor no son parte de HTML5, están muy relacionadas y definidas por el W3C

- Web Storage
- Web Workers
- Geolocation
- Canvas
  API para dibujar imágenes vectoriales 2D y *bitmap*.
  Hay librerías como jCanvas que facilitan su uso
- Web Messaging
  Para comunicación entre documentos de distintos orígenes, de manera razonablemente segura y sin las limitaciones de la *same-origin policy*
- y algunas otras
  `https://en.wikipedia.org/wiki/HTML5#New_APIs`

Web Storage

Web Storage es una API del W3C que permite almacenar información en el navegador en una forma que generalmente resulta más conveniente que las cookies

- Espacio reservado para este almacenamiento depende del navegador, normalmente entre 2.5Mb y 5 Mb por origen
  - Mucho más que las cookies, limitadas a 4 K
- La información de *Web Storage* la escribe el navegador y permanece en el navegador, solo se envía al servidor si se programa explícitamente
  - Las cookies las escribe el servidor y luego viajan en cada petición, consumen ancho de banda.

Por todo ello

- Si la información la necesita el cliente, *Web Storage* es mucho más conveniente
- Si la información la necesita el servidor, *Web Storage* no es tan útil, porque el servidor no tiene acceso directamente a ella

`https://stackoverflow.com/questions/3220660/local-storage-vs-cookies`

Web Storage es una estructura de diccionario: clave-valor
Incluye dos mecanismos:

- *sessionStorage*
  Un diccionario que dura tanto como la sesión. Se borra al cerrar el navegador
- *localStorage*
  Un diccionario persistente, el diccionario se mantiene aunque se cierre el navegador. Solo se borra si el usuario o la página lo borran explícitamente

Cada uno de estos diccionarios es distinto para cada origen (protocolo, host, puerto)

Almacenamiento de un valor

```
localStorage.setItem(clave) = valor;
sesionStorage.setItem(clave) = valor;
```

- El valor y la clave han de ser cadenas
- Si se intenta usar otro tipo de datos para el valor, muchos navegadores lo convierten a cadena
- Es preferible convertir el dato en JSON explícitamente

Recomendación:

- Normalmente un programa tendrá que almacenar varias variables. Nada impide guardar cada una de ellas por separado un una llamada a `setItem`
- Pero problemente es más conveniente almacenar todos esos valores como propiedades de un único *plain object*, y guardar ese objeto en una única llamada `setItem`

Recuperación de un valor

```
localStorage.getItem(clave);
sesionStorage.getItem(clave);
```

- Estos métodos devuelven el valor asociado a la clave
- O *undefined* si la clave no existe

Borrado de valores

```
localStorage.removeItem(clave)
sesionStorage.removeItem(clave)
```

- Borran la clave
- Devuelven *undefined*, tanto si la clave existía como si no

```
localStorage.clear()
sesionStorage.clear()
```

- Borran el diccionario completo del origen actual (protocolo, host, puerto)

Este programa pregunta al usuario su nombre solamente la primera vez que se ejecuta

```
'use strict'
let nombreUsuario = localStorage.getItem('nombreUsuario');
if (!nombreUsuario) {
  let input= prompt("¿Cómo te llamas?");
  localStorage.setItem('nombreUsuario',input);
} else {
  alert("Hola " + nombreUsuario);
}
for (let clave in localStorage) {
  let valor = localStorage[clave];
  console.log(clave+": "+valor)
}
```

http://ortuno.es/localStorage.html

Este programa borra el nombre de usuario, si estaba definido

```
let clave = 'nombreUsuario';
let valor=localStorage.getItem(clave);
if (valor){
    localStorage.removeItem(clave);
    alert(clave+ ' valía '+valor+ '. Ahora lo he borrado');
}else{
    alert(clave+" no definido");
};
```

http://ortuno.es/localStorage2.html

Referencias

- https://en.wikipedia.org/wiki/Web_storage
- https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
- https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API

Web Workers

Un *web worker* es un programa en JavaScript que se ejecuta en el navegador web en segundo plano, independiente de la página principal

- Es útil para realizar tareas intensivas en CPU
- Es útil para aprovechar los ordenadores multi núcleo (prácticamente todos los actuales)
- Se crea como instancia del objeto/la función `Worker()`, que recibe como argumento el nombre del script
- El web worker se comunica con el documento mediante el paso de mensajes: basta asignar el manejador a la propiedad *onmessage* del worker
- Se puede finalizar desde la página principal invocando al método `.terminate()` del worker
- Por motivos de seguridad, algunos navegadores como Chrome no permiten que se ejecuten en páginas cargadas localmente, solo en aquellas servidas desde un web

El siguiente ejemplo calcula dos números aleatorios, y cuando coinciden, envía un mensaje

```
'use strict'
function random(x){
    return Math.floor((Math.random() * x) + 1);
}

let tamanio=1000000;
let x,y;
let c=100;
while (c>0){
    x=random(tamanio);
    y=random(tamanio);
    if (x===y) {
          postMessage(x);
          c-=1;
    }
}
```

http://ortuno.es/web_worker.js.txt

```
// Chrome no permite ejecutar un web worker localmente,
// tiene que estar servido desde web
'use strict'
let w;

let texto_salida = document.querySelector("#texto_salida");

if (typeof(Worker) !== "undefined") {
  if (typeof(w) == "undefined") {
    w = new Worker("js/web_worker.js");
  }
  w.onmessage = function(event) {
    texto_salida.textContent = event.data;
  };
} else {
  console.log("Web workers no soportados");
}
```

http://ortuno.es/web_worker.html

Geolocation

Los navegadores modernos pueden conocer, si el usuario lo permite, su ubicación geográfica
Posiblemente el más preciso es Google Chrome

- En dispositivos con GPS, la información proviene del GPS
- En conexiones cableadas, obtiene información a partir de la dirección IP
- En conexiones WiFi
  - Obtiene las MAC de los Access Point WiFi colindantes
  - La compara con la base de datos de Google de la posición de los Access Point
- En conexiones de telefonía móvil
  - Compara el identificador de la celda con su base de datos de posiciones de celdas de telefonía

¿Cómo obtiene Google la base de datos de Access Point y celdas de telefonía móvil?

- Con los coches que capturan datos para Google Maps. (Aunque esto le causó problemas legales)
- Con la información de los millones de dispositivos Android con GPS

Para aceder a las coordenadas basta usar el método `navigator.geolocation.getCurrentPosition()` pasándole como parámetro

- La función que procesará las coordenadas
- La función que se invocará en caso de error
- Un objeto con opciones

```
let options = {
  enableHighAccuracy: true,
  timeout: 5000,
  maximumAge: 0
};
function success(pos) {
  let x = pos.coords;
  let mensaje = 'Posición actual\n';
  mensaje += 'Latitud :' + x.latitude;
  mensaje += '\nLongitud :' + x.longitude;
  mensaje += '\nPrecisión :' + x.accuracy + " metros";
  alert(mensaje);
}
function error(err) {
  console.warn(`ERROR(${err.code}): ${err.message}`);
};
navigator.geolocation.getCurrentPosition(success, error, options);
```

http://ortuno.es/geoloc.html

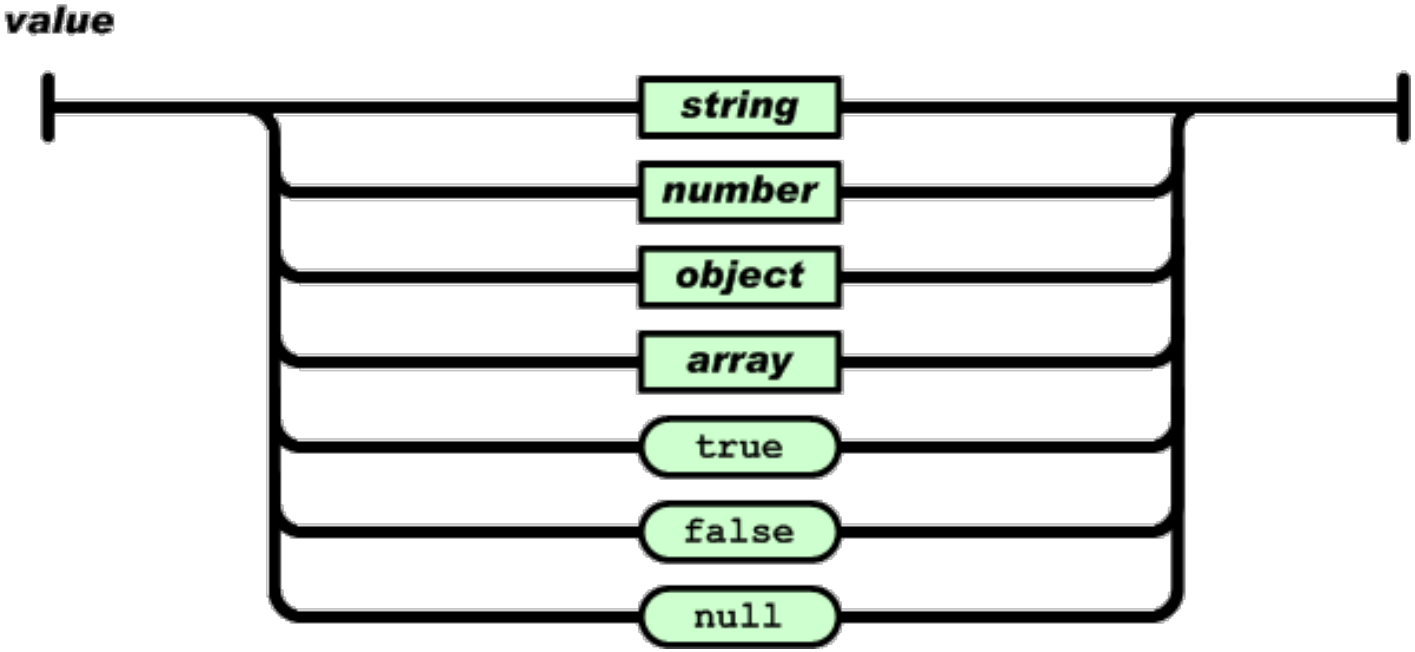
Json

JSON

Es un formato ligero para intercambiar datos, independiente del lenguaje de programación y de la plataforma

- Los datos se codifican sobre una cadena de texto utf-8
- Estándar abierto, RFC 4627, año 2006
- Originalmente se consideraba subconjunto del lenguaje JavaScript y se denominaba *JavaScript Object Notation*, aunque ya no es parte de JavaScript
- Diseñado como alternativa a XML, más ligero. Actualmente es más popular que XML
- Carece de algunas características de XML, por ejemplo gramáticas o diferencia entre texto y metadato

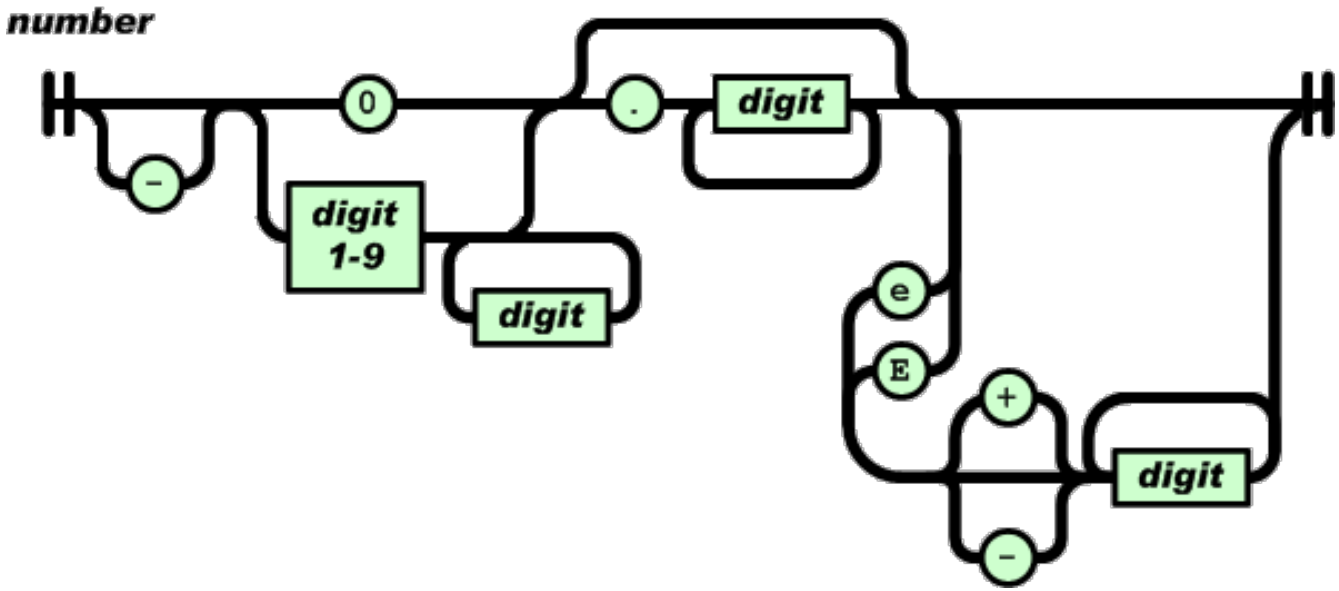
Value



Fuente:json.org

- Un valor JSON puede ser un número, una cadena, un array o un objeto, además de las constantes *true*, *false* y *null*
- Igual que JavaScript. Excepto que *undefined* no es un valor JSON

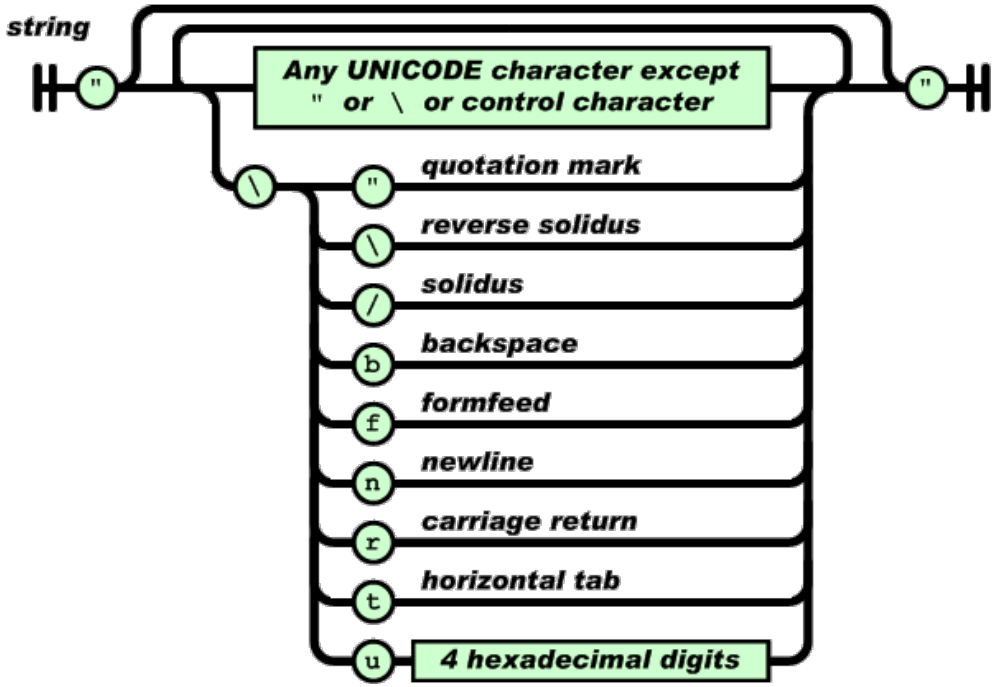
Number



Fuente:json.org

- Los números son como las constantes numéricas de cualquier lenguaje de programación moderno

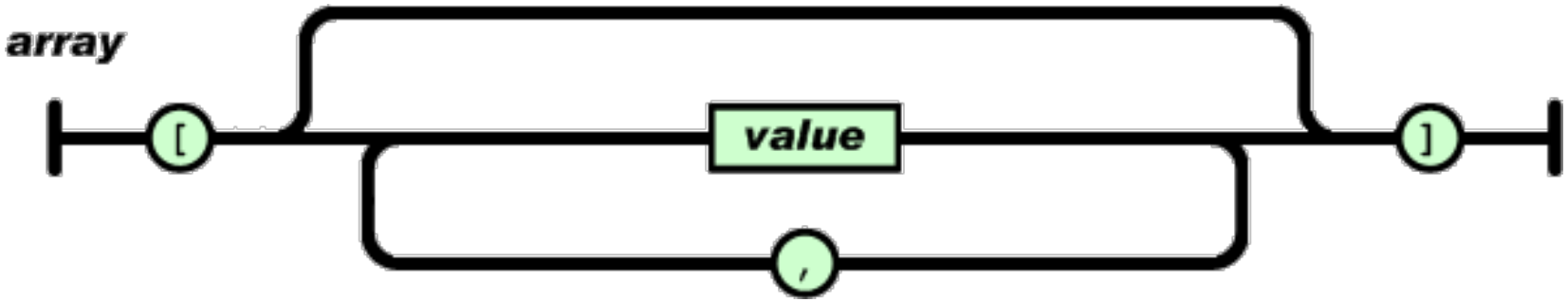
String



Fuente:json.org

- Las cadenas son como las de cualquier lenguaje de programación moderno
- El delimitador es la comilla doble
  - JavaScript admite la comilla doble y la simple, aunque la más habitual es la simple

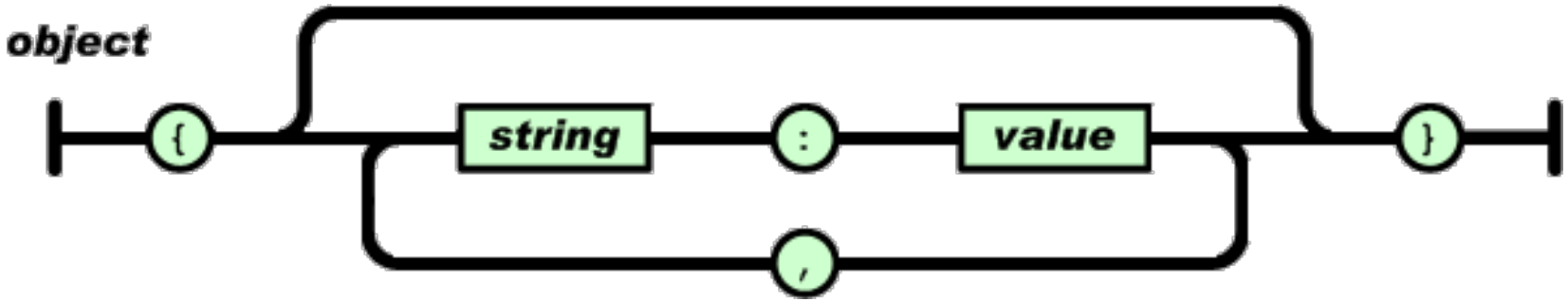
Array



Fuente:json.org

- Un array es una secuencia de valores entre corchetes, separados por comas
- Igual que JavaScript

Objetos



Fuente:json.org

- Un objeto es una secuencia de pares clave:valor, separados por comas
- Las claves son cadenas
- Igual que JavaScript

Ejemplos correctos

- `"hola, mundo"`
- `4243.12`
- `-947e-5`
- `null`
- `[1,2,3,4]`

- `[1, "azul", [1,2,3]]`

- ```
  [
      1,
      "azul",
      [
          1,
          2,
          3
      ]
  ]
  ```

- `["as" ,  "dos", "tres"]`

- `["sota", "caballo", "rey"]`

- ```
  [
      "sota",
      "caballo",
      "rey"
  ]
  ```

- `{ "nombre":"Juan", "apellido":"Pérez"}`
- `{ "v1":true, "v2":null, "v3":false}`
- `{ "nombre": "Juan", "notas":[5.5, 7.2, 6.1]}`
-
```
{
    "nombre": "Juan",
    "notas": [
        5.5,
        7.2,
        6.1
    ]
}
```

Ejemplos incorrectos

- `True`
- `'hola, mundo'`
- `{"hola,mundo"}`
- `{1:"uno", 2:"dos"}`

Conversión de objeto en cadena JSON

En JavaScript, para convertir un objeto en una cadena JSON usamos la *built-in function* JSON.stringify

```
'use strict'
let lista=[ "sota", "caballo", "rey" ];
console.log(typeof(lista),lista);
// object ["sota","caballo","rey"]

let cadena=JSON.stringify(lista);
console.log(typeof(cadena),cadena);
// string ["sota","caballo","rey"]
```

Conversión de cadena JSON en objeto

Para convertir una cadena de texto con un JSON en un objeto JavaScript, disponemos de la *built-in function* JSON.parse

```
'use strict'
let cadena='{ "nombre":"redes", "curso":1,
    "horario":["L1500", "X1700"] }'
console.log(typeof(cadena),cadena);
// string
//{"nombre":"redes", "curso":1,  "horario":["L1500", "X1700"] }

let objeto=JSON.parse(cadena);
console.log(typeof(objeto),objeto);
// object
//{ nombre: 'redes', curso: 1,  horario: [ 'L1500', 'X1700' ] }
```

Si la cadena no cumple el formato JSON, se genera una excepción *SyntaxError*

AJAX

Ajax

Es un conjunto de técnicas para enviar información asíncrona entre servidor web y cliente

- Aunque hay antecedentes desde 1996, el Ajax actual lo crea google en 2004, para gmail y google maps
- Evita que el usuario necesite pulsar *enviar*. La aplicación web recibe información continuamente, resultando una experiencia de usuario similar a la de una aplicación de escritorio
- Originalmente significaba Asynchronous JavaScript and XML, pero cuando empieza a usarse sin JavaScript y sin XML, el acrónimo AJAX se abandona para usar la palabra Ajax

Funcionamiento de Ajax

1. El usuario solicita una URL desde su navegador
2. El servidor web devuelve la página, que contiene un script
3. El navegador presenta la página y ejecuta el script
4. El script hace peticiones HTTP asíncronas a una URI del servidor, sin que el usuario intervenga. El servidor suele ser RESTful
5. El script interpreta las respuestas HTTP y modifica el HTML, sin que el usuario intervenga

Same-origin policy

Same-origin policy es una norma que aparece en Netscape 2 (año 1995), que se ha convertido en un estándar. Consiste en que el código JavaScript solo puede acceder a datos que provengan del mismo origen desde el que se ha cargado el script

- Se entiende por *origen* el mismo protocolo, máquina y puerto

Ejemplo

- El usuario accede a una página web en *molamazo.com*,
- Esta página web puede tener código JavaScript que acceda a datos que estén en *molamazo.com*, pero solamente en este sitio
- No puede acceder a datos en *bancofuenla.es*
  - De lo contrario, una vez que el usuario se autentica en *bancofuenla* con una página de *bancofuenla*, un script malicioso en *molamazo* podría acceder a información sensible en *bancofuenla*

JSOP

JSONP (*JSON with padding, JSON con relleno*) es una técnica que permite que una página web obtenga datos desde un sitio web distinto al suyo, sin vulnerar la *same-origin policy*

- Es un protocolo del año 2005, soluciona el problema pero no es especialmente elegante. También tiene algunos problemas de seguridad potenciales
- Una alternativa más avanzada pero menos extendida es CORS, *Cross-origin resource sharing*

JSONP requiere de la colaboración del servidor. Es necesario que un servidor ofrezca datos no solo en JSON, también en JSONP

JSONP se basa en que la *same-origin policy* se aplica solo a datos, no a scripts

- Ejemplo: un script descargado desde *molamazo.com*, no puede descargar datos desde *bancofuenla.es*, pero sí puede descargar otro script
- Se entiende que esto no es especialmente peligroso.
  Bancofuenla podría enviar una contraseña a una página web, como un dato, esperando que la lea el usuario.
  Pero en un script nunca debería haber información sensible

Ejemplo de uso de JSOP

- Enviar 3 no está permitido, es un dato.
- Pero la cadena `"f(3)"` no es un dato. Es un script. Se puede enviar.
  - Si *bancofuenla* accede a enviar `"f(3)"` como dato JSON, sabe que esta información podría ser usada por un script de cualquier otro sitio, p.e. *molamazo*. Por tanto, nunca enviará de este modo información sensible
  - Un problema distinto, que JSONP no contempla, es que sea *bancofuenla* quien aproveche esto para enviar código dañino

En JSONP, el cliente solicita un dato a un sitio web (en nuestro ejemplo, *bancofuenla*), y también le indica el nombre de la función en la que se *envuelve* el dato.

- Esto se hace con un parámetro que suele denominarse `callback`
  Una petición podría ser
  `bancofuenla.es/divisas.html?par=USDEUR&fecha=hoy&callback=procesaDivisas`

- A la que el servidor podría responder
  `procesaDivisas(0.865)`
- `procesaDivisas()` será una función en el script cliente, que tratará el dato (0.865)

Ajax con jQuery

jQuery facilita mucho el uso de Ajax. El método principal es `ajax()`
Recibe:

- Un *plain object* que puede tener varios atributos. El principal es `url`, la dirección del servicio

Devuelve:

- Un objeto de tipo `jqXHR`, que tiene varios atributos. Los principales son:
  - Un método llamado `done` al que le pasamos la función que se invocará si la petición tiene éxito. Tendrá un parámetro, que en nuestro caso será un objeto JSON
  - Un método llamado `fail` al que le pasamos la función a invocar en caso de error

El siguiente ejemplo llama a `http://fixer.io`, un servicio que ofrece tipos de cambio de divisas

- Por motivos de seguridad, para cumplir con la *same-origin policy*, solo podemos cargar la página desde el disco duro local, no desde el web
- Si este servicio ofreciera respuestas en JSONP, sí podríamos usarlo normalmente. jQuery se ocupa del manejo de JSONP, resulta transparente para el programador
- Otra solución para los sitios sin soporte de JSONP (que son muchos), sería un proxy en el mismo sitio web que sirve el HTML

```
$(document).ready(function() {
  let urlServicio = 'http://data.fixer.io/latest';
  peticion = $.ajax({
    url: urlServicio,
    data: {
      access_key: "xxxxxx",
      symbols: "USD, GBP"
    }
  })
  peticion.done(manejaRespuesta);
  peticion.fail(manejaError);

  function manejaRespuesta(json) {
    $("#div01").text(JSON.stringify(json));
  };

  function manejaError(jqXHR) {
    $("#div01").text("Error: " + jqXHR.status);
  };
});
```

Cambio de divisas

- Cada divisa tiene un código ISO 4217, que es un identificador de 3 letras mayúsculas. Por ejemplo USD (United States Dollar) para el dólar, EUR para el euro, GBP para la libra esterlina, etc
- El precio de una divisa frente a otra se indica mediante la concatenación de sus dos códigos ISO 4217. La primera se denomina *divisa base* y la segunda, *divisa cotizada*
- Ejemplo EURUSD $= 1.13$
  Aquí la divisa base es el euro, y la cotizada, el dólar.
  Este valor (1.13) indica cuántas unidades de la divisa cotizada hacen falta para comprar una unidad de la divisa base
- Dicho de otro modo, la primera moneda es la que queremos y la segunda, la que tenemos, la que usamos para pagar.

setInterval

Para que la consulta Ajax se repita periódicamente, podemos usar la función `setInterval()`
Recibe

- Como primer argumento, una función
- Como segundo argumento, un intervalo de tiempo en milisegundos

Como resultado, evalúa la función periódicamente, con el intervalo especificado. El siguente ejemplo se ejecutaría cada 60 segundos

```
setInterval(function() {
                miTexto=actualizaTexto();
                $("#p01").text(miTexto);
            },
            60000
);
```

Ejemplos de API

YouTube

Insertar un reproductor de vídeo de YouTube en JavaScript es muy sencillo

- En el cuerpo del HTML incluimos un div con un identificador, donde irá el reproductor
- Cargamos el script `https://www.youtube.com/iframe_api`
- En la función `onYouTubeIframeAPIReady()`, creamos una instancia de YT.Player, y en sus atributos fijamos el tamaño del reproductor y el identificador del video
- En la función `onPlayerReady()`, invocamos `event.target.playVideo()`

```
// Carga asíncrona del API del IFrame Player de YouTube
let tag = document.createElement('script');
tag.src = "https://www.youtube.com/iframe_api";
let firstScriptTag = document.getElementsByTagName('script')[0];
firstScriptTag.parentNode.insertBefore(tag, firstScriptTag);

// Creación de un iframe, con reproductor de YouTube
let player;
function onYouTubeIframeAPIReady() {
  player = new YT.Player('mi_reproductor', {
    height: '768',
    width: '1024',
    videoId: 'lKCCZTUx0sI',
    events: {
      'onReady': onPlayerReady,
    }
  });
}

// El API llama a esta función cuando el reproductor esté listo
function onPlayerReady(event) {
  event.target.playVideo();
}
```

http://ortuno.es/youtube.html

OpenStreeMap

OpenStreetMap (OSM) es un proyecto que tiene como objetivo crear un mapa del mundo editable y libre

- Creado por Steve Coast en el Reino Unido en el año 2004
- Muy similar a Wikipedia, por su filosofía y funcionamiento
- Muy similar a Google Maps en cuanto a funcionalidad y calidad
- Hay mucho software y muchas APIs relacionadas con este proyecto: captura de datos, generación de mapas, de capas, etc
- El uso básico consiste en integrar un mapa y puntos de interés en un página HTML. Una de las APIs más populares para esto es leaflet (`http://leafletjs.com`)

La representación de mapas se basa en *tiles* (azulejos, baldosas)

- Un *tile* es un *bitmap* / mapa de bits con un fragmento de mapa. Su tamaño típico es 256x256 pixeles. Aunque en ocasiones se usa 64x64 (para dispositivos móviles) o 512x512, para dispositivos de alta resolución
- Los datos de OpenStreetMap son libres (y gratuitos). El servidor de *tiles* no lo es
  - Podemos contratarlo o desplegar uno propio
  - Para usos experimentales, con poca carga de datos, podemos usar alguno proporcionado por la propia organización, como `tile.osm.org`

Representación de un mapa OSM con leaflet

Para representar un mapa OpenStreetMap en una página HTML, usando la librería leaflet

- Incluimos la librería desde el CDN

```
<script src="https://unpkg.com/leaflet@1.2.0/dist/leaflet.js">
```

  Esto crea un objeto global llamado L
- En nuestro HTML definimos un DIV donde irá el mapa. Es necesario que tenga definido un atributo CSS *height* con la altura

```
  <style>
    #id_mapa {
      height: 400px;
    }
  </style>

...

  <div id="id_mapa"></div>
```

- Creamos un objeto de tipo `L.map()`, pasando las coordenadas del centro del mapa deseado y el zoom inicial
  (el zoom 0 es un mapa de toda la tierra, numeros mayores van haciendo el mapa más detallado)
- Invocamos el método `L.tileLayer()`, con la URL del mapa en el servidor de tiles y el mensaje de atribución del mapa

```
$(document).ready(function() {
    let latitud=40.417; //coordenada y
    let longitud=-3.703; //coordenada x
    let zoom=16;
    let mi_mapa = L.map('id_mapa').setView([latitud, longitud], zoom);
    L.tileLayer('http://{s}.tile.osm.org/{z}/{x}/{y}.png', {
        attribution:\
        '&copy; <a href="http://osm.org/copyright">OpenStreetMap</a>'
    }).addTo(mi_mapa);
  });
```

http://ortuno.es/openstreet.html

Marcadores y mensajes popup

- Podemos añadir marcadores, invicando el método `L.marker(<COORDENADAS>).addTo(<OBJETO MAP)`
- Podemos añadir mensajes emergentes de tipo *popup*, con el método `.bindPopup` de un marcador

```
let mi_marcador = L.marker(coord_labIII).addTo(mi_mapa);
mi_marcador.bindPopup("Laboratorios III").openPopup();
```

http://ortuno.es/openstreet2.html

Express

Express.js

Express.js, también llamado simplemente *Express* es un *Web Application Framework*

- Permite desarrollar aplicaciones web en el servidor usando el mismo lenguaje que en el cliente: JavaScript
- Basado en Node.js
- Alternativa a Django o Ruby on Rails
- Aplicación libre y gratuita, muy popular
- Desarrollado por TJ Holowaychuk en 2010. Vendido a StrongLoop, que actualmente pertenece a IBM

Instalación de Express

- Para usar Express.js necesitamos una versión reciente de node.js. Con Ubuntu 20.04 se distribuye node.js v10, que no cumple este requisito
- Para saber qué versión de node.js tenemos instalada:

```
node -v
```

- Para instalar node.js v16 en Ubuntu 20.04 (si tenemos privilegios de root)

```
cd
curl -sL https://deb.nodesource.com/setup_16.x | sudo bash -
sudo apt-get install -y nodejs
```

- Para instalar express en macOS o Linux (no son necesarios privilegios de root)

```
npm install express
```

- Para saber qué versión de express.js tenemos instalada

```
npm list express
```

Diseño de las URL

El interface de una aplicación (las URL que usarán cliente y servidor) en Express.js o en cualquier otra herramienta, debe seguir los mismos princios de calidad. Una URL bien diseñada debería permanecer un número indefinido de años (*toda la vida*), sin importar que cambien las tecnologías: framework, lenguaje, sistema operativo etc. Para ello:

- No exponer detalles técnicos. No se debería ver palabras como *php asp* o incluso *js*
  - Bien:
    http://www.ejemplo.com/busqueda
  - Mal:
    http://www.ejemplo.com/busqueda.php

- Evitar palabras irrelevantes. Una URL debe ser corta, cada palabra debe tener significado. P.e. si todas las URL empiezan por */home/*, esa palabra sobra
  - Bien:
    http://www.ejemplo.com/busqueda
  - Mal:
    http://www.ejemplo.com/home/app/auto/busqueda

- Ser consistente con la separación de palabras. Se pueden usar guiones, barras bajas, NotacionCamello o notacionDromedario, con tal de que siempre se use la misma. Generalmente se recomiendan los guiones y todas las palabras en minúsculas, aunque es opinable.
  - Bien:
    http://www.ejemplo.com/user-id/:user-id/app-id/:app-id
    http://www.ejemplo.com/userId/:userId/appId/:appId
  - Mal:
    http://www.ejemplo.com/userId/:userId/app-id/:app-id
- No usar nunca espacios ni caracteres no ingleses
  - Bien:
    http://www.ejemplo.com/spain
  - Mal:
    http://www.ejemplo.com/españa

APIs REST con Express.js

- Con Express.js podemos preparar de forma muy sencilla servicios web con interface REST / ROA
- Esencialmente consiste en un servidor web que responde peticiones GET, PUT, POST y DELETE siguiendo la metodología REST

`http://ortuno.es/rest.pdf`

- Las más habituales son las peticiones GET: el cliente web solicita un recurso al servidor, indicando su URL. Normalmente será un fichero generado dinámicamente (por una aplicación)
- Para actualización de datos, mediante el método POST el cliente web envía datos al servidor en el cuerpo de la petición. Estos datos serán usados por una aplicación en el servidor

Configuración de Express

Para lanzar Express, preparamos un fichero `.js` y lo ejecutamos con node. Ejemplo:

```
node api_rest.js
```

Atención, si el puerto ya está ocupado (típicamente porque, por error, hemos lanzado un servidor sin cerrar el anterior) el mensaje no es demasiado claro

```
events.js:174
      throw er; // Unhandled 'error' event
```

Para probar el servidor

- Podemos usar *Postman API platform*, una herramienta libre y gratuita, muy popular
- Podemos user *RESTer*, una extensión para Chrome
- Las peticiones GET podemos probarlas, además, desde cualquier navegador web

Objeto app

Las dos primeras líneas de un programa en express suelen ser

```
const express = require('express');
//  Importamos el módulo express, que por omisión
//  exporta una función

const app = express();
//  Asignamos esa función al objeto app
```

En el resto del programa, invocamos a métodos como

`app.get()  app.put()`

Esto es algo normal en JavaScript pero raro en otros lenguajes: una función es un caso particular de objeto, que a su vez puede tener métodos

Routing

- Una vez listo el objeto app, nos ocupamos del *routing*, esto es, para cada petición del cliente, indicar qué respuesta se le dará
- Con los métodos `get`, `put`, `post`, `delete` indicamos qué hacer con las peticiones GET, PUT, POST, DELETE
  - El primer parámetro indicamos la URL
  - El segundo parámetro es el manejador de la petición, la función que se disparará cuando llegue una petición a la URL

```
app.get('/about', (req, res) => {
  res.type('text/plain; charset=utf-8');
  res.send('Esto es una prueba de Express');
})
```

Se suele usar *notación flecha*, pero nada impide emplear notación tradicional

El manejador tiene dos parámetros, normalmente llamados

1. `req`
   Objeto con todos los detalles de la petición (*request*)
2. `res`
   Objeto con todos los detalles de la respuesta (*response*)
   - `res.send()` permite responder texto, indicando previamente la codificación con `res.type()`
   - `res.json()` seponde al cliente un objeto json

Parámetros

Podemos usar *parámetros* en la dirección: segmentos de la URL que capturan los valores especificados en esa posición. Se indican anteponiendo el carácter *dos puntos*. Los parámetros se guardan en el objeto *params* del objeto *req*. Ejemplo:

```
app.get('/api/coords/:x/:y', (req, res) => {
  let x = req.params.x
  let y = req.params.y
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido las coordenadas '+ x + ' ' + y);
})
```

El cliente podrá hacer una petición GET a la dirección `/api/coords/150/300`, y capturaremos los valores 150 y 300 para x e y, respectivamente

No hay inconveniente en alternar parámetros con segmentos fijos
Ej:

```
/user/:userId/subject/:subjectId
```

Para nombrar estos parámetros podemos usar letras inglesas mayúsculas o minúsculas, números y la barra baja

Peticiones PUT

Para acceder al cuerpo de una petición PUT, ejecutamos `app.use(express.json())` [12]. El método *use* permite añadir capas de *middleware*, esto es, código intermedio que procesa todas las peticiones

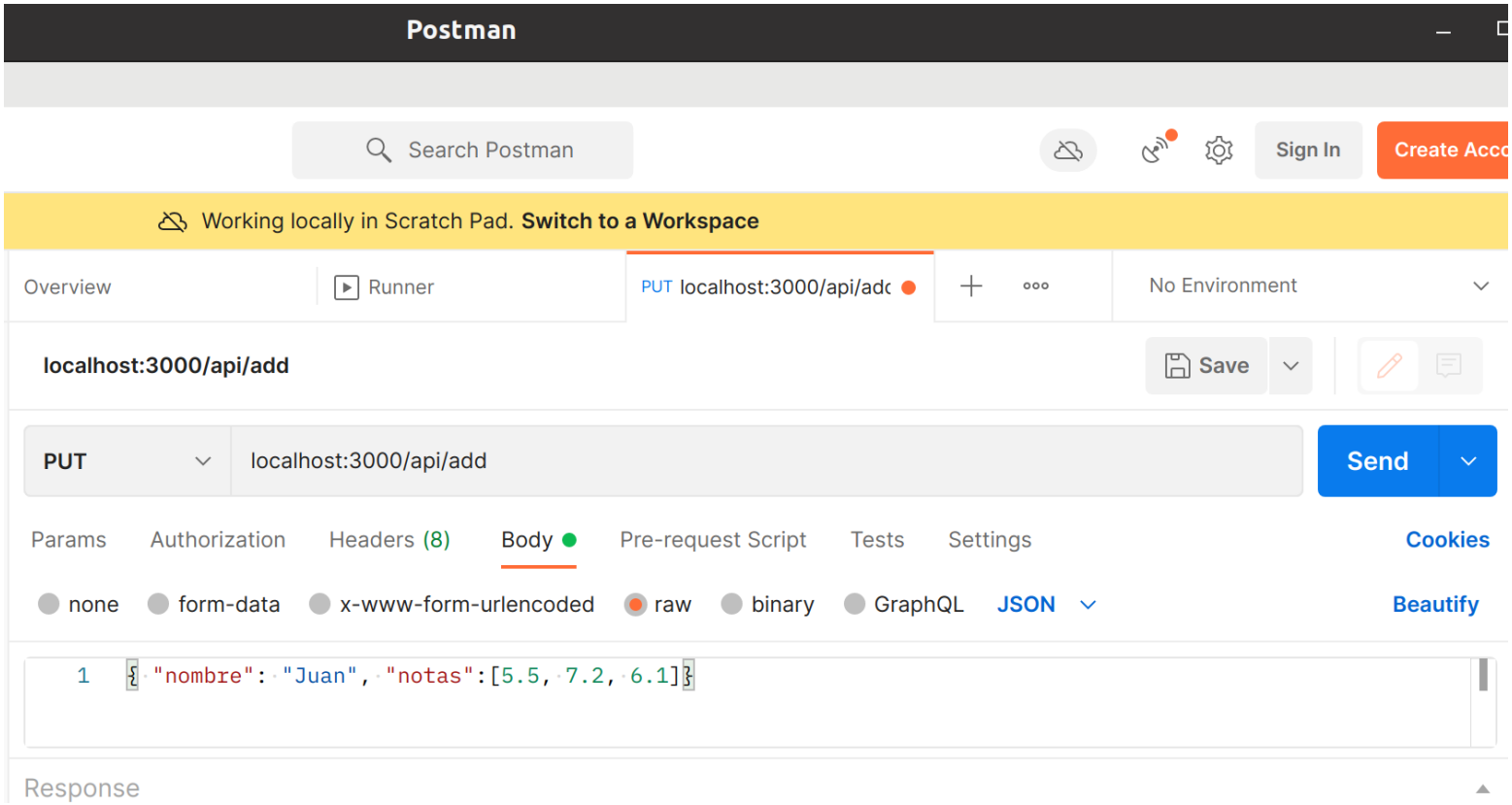
```
app.use(express.json());
[...]
app.put('/api/add',(req,res) => {
  console.log('Me has enviado este objeto:');
  console.log(req.body);
  res.json(req.body);
});
```

- El cuerpo de la petición está disponible en `req.body`
- Será un objeto JSON (no un valor cualquiera sino un objeto: una secuencia de pares clave-valor, entre llaves)

[12] La documentacióna antigua indica que es necesario instalar la librería body-parser, pero esto es obsoleto desde express 4.16 (año 2017)

postman

- Para probar las peticiones PUT, necesitamos una herramienta como *postman*
- Lo podemos instalar con
  `snap install postman`
- En el laboratorio de la ETSIT ya está instalado, en `/opt/Postman/postman`
- Para el uso básico de postman, no es necesario crear ninguna cuenta



- Elegimos `PUT` y escribimos la URL
- En el *body* escribimos una petición de tipo *raw* en JSON, con el objeto que corresponda
- Pulsamos *send*

Errores

Despues del *routing* de las peticiones previstas, añadimos los manejadores de los errores. Es importante hacerlo en este orden, para que se ejecuten solamente cuando no encaje ninguna ruta especificada previamente

```
//  Status Code 404
app.use((req, res) => {
  res.type('text/plain');
  res.status(404);
  res.send('404 - Not Found');
})

// Status Code 500
app.use((err, req, res, next) => {
  console.error(err.message);
  res.type('text/plain');
  res.status(500);
  res.send('500 - Server Error');
})
```

Status Code

Como en cualquier servidor web, cada petición a express.js debe recibir un *status code*, con los criterios habituales: error del servidor, error del cliente, petición sin errores

- Error de servidor catastrófico
  Error severo en el servidor, típicamente excepción sin manejar. Requieren iniciar el servidor. `Status code 500`
- Error de servidor recuperable
  Error en el servidor que tal vez no sea permanente, como fallo no previsto en un fichero o en la base de datos.
  `Status code 500`

- Errores del cliente
  El cliente usa mal el API: pide un recurso que no existe o no tiene los permisos adecuados. Desde el punto de vista del servidor, esto no es un error, es un comportamiento normal.
  Códigos más habituales:
  `404 (Not Found)`, `400 (Bad Request)`, `401 (Unauthorized)`
- Sin errores
  Cuando la petición es correcta, el servidor devolverá `Status Code 200`.
  - Si el cliente pide una lista de elementos, y la lista está vacía, esto también es un resultado correcto, no un error

Método listen()

Una vez configurado el *routing()*, ejecutamos `app.listen()` pasando dos argumentos

- Puerto TCP donde el servidor aceptará las peticiones
- Función a ejecutar en el servidor, típicamente para escribir mensaje en salida estándar

```
const puerto = process.env.PORT || 3000;
// Puerto indicado en la variable de entorno PORT, o 3000
// si la variable no está definida.

[...]

app.listen(puerto, () => console.log(
  `Express iniciado en http://localhost:${puerto}\n` +
  `Ctrl-C para finalizar.`));
```

Ejemplo completo

```
const express = require('express');
// Importamos el módulo express, que exporta una función

const app = express();
// Función principal, que a su vez tiene métodos

const puerto = process.env.PORT || 3000;
// Puerto indicado en la variable de entorno PORT, o 3000
// si la variable no está definida.

app.use(express.json());
// middleware necesario para procesar las peticiones
// POST que incluyan un cuerpo json
```

```
// Direccionamiento para GET
app.get('/', (req, res) => {
  res.type('text/plain; charset=utf-8');
  res.send('Bienvenidos a mi página de ejemplo');
})

app.get('/about', (req, res) => {
  res.type('text/plain; charset=utf-8');
  res.send('Esto es una prueba de Express');
})

app.get('/data', (req, res) => {
  res.json("['sota', 'caballo', 'rey']");
})
```

```
app.get('/api/carta/:id', (req, res) => {
  let id = req.params.id
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido la carta '+id);
})

app.get('/api/coords/:x/:y', (req, res) => {
  let x = req.params.x
  let y = req.params.y
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido las coordenadas '+ x + ' ' + y);
})
```

```
// Direccionamiento para PUT
app.put('/api/object/:id', (req, res) => {
  let id = req.params.id
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido crear el objeto '+ id );
})

app.put('/api/add',(req,res) => {
  console.log('Me has enviado este objeto:');
  console.log(req.body);
  res.json(req.body);
});
```

```
// Esta llamada debe ser la última, error 404
app.use((req, res) => {
  res.type('text/plain');
  res.status(404);
  res.send('404 - Not Found');
})

// custom 500 page
app.use((err, req, res, next) => {
  console.error(err.message);
  res.type('text/plain');
  res.status(500);
  res.send('500 - Server Error');
})

app.listen(puerto, () => console.log(
  `Express iniciado en http://localhost:${puerto}\n` +
  `Ctrl-C para finalizar.`));
```

http://ortuno.es/api_rest.js.html

Cómo servir ficheros estáticos

- Aunque el propósito principal de Express.js es servir páginas web generadas dinámicamente, en ocasiones necesitaremos servir ficheros *estáticos*, esto es, páginas web que se correspondan con ficheros en el servidor *tal cual*
- Para esto, es fundamental tener claro qué significa *directorio raiz* de un sitio web, sin confundirlo con el directorio raiz del sistema de ficheros (disco duro) del ordenador que sirve el web

Directorio raiz de un sitio web

- El *directorio raiz* de un sitio web es el directorio que los clientes web percibirán como el directorio /
- Se corresponderá con cierto directorio en el servidor web, al que llamaremos así, *directorio raiz*. P.e. `/var/www/html/`
- No debemos confundirlo con el directorio raiz del sistema de ficheros (disco duro) del servidor web (/)
  - Que naturalmente será inaccesible via web, a menos que un atacante explote un fallo de seguridad muy severo

Ejemplo 1

Si el servidor web está en el puerto 3000 de `localhost` y `dir_raiz` vale

```
/home/jperez/www/site01
```

Y el el servidor web tiene el fichero

```
/home/jperez/www/site01/holamundo.html
```

El cliente deberá pedirlo como

```
localhost:3000/holamundo.html
```

Observa que el nombre del directorio raiz no forma parte del path que debe pedir el cliente web

Ejemplo 2

Si el servidor está en

```
localhost:3000
```

y `dir_raiz` vale

```
/home/jperez/www
```

Y el el servidor tiene el fichero

```
/home/jperez/www/site01/holamundo.html
```

El cliente deberá pedirlo como

```
localhost:3000/site01/holamundo.html
```

Especificación del home

Muy importante: recuerda que (en ningún lenguaje) deberías escribir el directorio *home* como una cadena literal (*escribirla tal cual*, p.e. `/home/jperez`), sino leerlo desde la variable de entorno *home*

Al directoro *home* se le suele llamar de distintas formas, como

- `$HOME`
  Esta sintaxis es válida en la shell de linux, pero en ningún otro entorno
- `~/`
  Esta sintaxis es válida en la shell de linux y en algunas librerías de algunos lenguajes, pero raramente se puede escribir *tal cual* en un lenguaje de programación

En node.js, y por tanto en express.js, para construir el directorio

```
~/www/site01
```

Escribiríamos

```
path.join( process.env.HOME, "www/site01");
```

Esto generará el directorio que corresponda a nuestro usuario, nuestra máquina y nuestro sistema operativo
P.e. una cuenta de nuestro laboratorio, de nuestra máquina Linux o nuestra máquina macOS podría ser, respectivamente:

```
/home/alumnos/agarcia/www/site01
/home/jperez/www/site01
/Users/Ana
```

En este caso, no hay diferencia entre escribir

```
path.join( process.env.HOME, "www/site01");
```

o

```
path.join( process.env.HOME, "/www/site01");
```

Ya que

- El método `path.join()` concatena dos trayectos (sin importar si el último acaba en barra o no, o el primero empieza por barra o no)
- El segundo argumento de *join* no es ni un trayecto relativo ni un trayecto absoluto, sino un trayecto a concatenar con el trayecto especificado en el primer argumento

- Esto es lo que recomendamos aquí: especificar siempre el directorio raiz de un sitio web a partir de variables de entorno (ya sea *HOME*, como acabamos de ver, ya sea alguna otra variable que definamos en nuestro `~/.bashrc` o similar)
- Pero verás muchos libros y tutoriales que usan trayectos relativos, p.e

  ```
  app.use(express.static('public');
  ```

  Esto significa *directorio llamado public, contenido dentro del directorio actual del proceso que hizo la llamada a express*.
  - Es perfectamente válido, aunque suele resultar muy confuso para el principiante

Para servir los ficheros estáticos de un directorio, *tal cual*, basta con

```
app.use(express.static(dir_raiz))
```

Si queremos servir más de un directorio, hacemos varias llamadas a este método

```
app.use(express.static(dir_public))
app.use(express.static(dir_js))
```

Naturalmente, los subdirectorios están siempre incluidos, recursivamente

Ficheros estáticos, ejemplo completo

```
const express = require('express');
const app = express();
const puerto = process.env.PORT || 3000;

const path = require('path');
// Importamos el módulo path

dir_raiz = path.join( process.env.HOME, "www/site01");
// Construye ~/www/site01

app.use(express.static(dir_raiz))
// Sirve todos los ficheros del directorio raiz

// Ahora haríamos las llamadas a app.use para tratar los
// errores 404 y 500, así como la llamada a app.listen(),
// de la forma habitual.
```

http://ortuno.es/estaticos01.js.html

En ocasiones querremos que cuando el cliente pida cierto fichero, reciba otro distinto. P.e, por omisión, / apunta a `index.tml`. Si quisiéramos que cuando pida / reciba `holamundo.html`

```
app.get('/', (req, res) => {
    res.sendFile(path.join(dir_raiz,'holamundo.html'));
});
// para '/', sirve ~/www/site01/holamundo.html

app.use(express.static(dir_raiz))
// Sirve todos los ficheros del directorio raiz
```

Recuerda que es importante el orden en que se llama a `app.use()`, la primera invocada tiene prioridad

http://ortuno.es/estaticos02.js.html

Clientes REST en JavaScript

Introducción a los clientes REST en JavaScript

Hasta ahora hemos visto

- Cómo ejecutar programas JavaScript dentro del navegador
- Cómo servir programas y ficheros con interface REST/ROA en el servidor, usando Express.js

A continuación, veremos cómo programar un cliente REST/ROA en JavaScript dentro del navegador. Lo probaremos con un servidor en Express.js

Same-origin policy

Same-origin policy es una norma que aparece en Netscape 2 (año 1995), que se ha convertido en un estándar. Consiste en que el código JavaScript solo puede acceder a datos que provengan del mismo origen desde el que se ha cargado el script

- Se entiende por *origen* el mismo protocolo, máquina y puerto

Ejemplo

- El usuario accede a una página web en *molamazo.com*,
- Esta página web puede tener código JavaScript que acceda a datos que estén en *molamazo.com*, pero solamente en este sitio
- No puede acceder a datos en *bancofuenla.es*
  - De lo contrario, una vez que el usuario se autentica en *bancofuenla* con una página de *bancofuenla*, un script malicioso en *molamazo* podría acceder a información sensible en *bancofuenla*

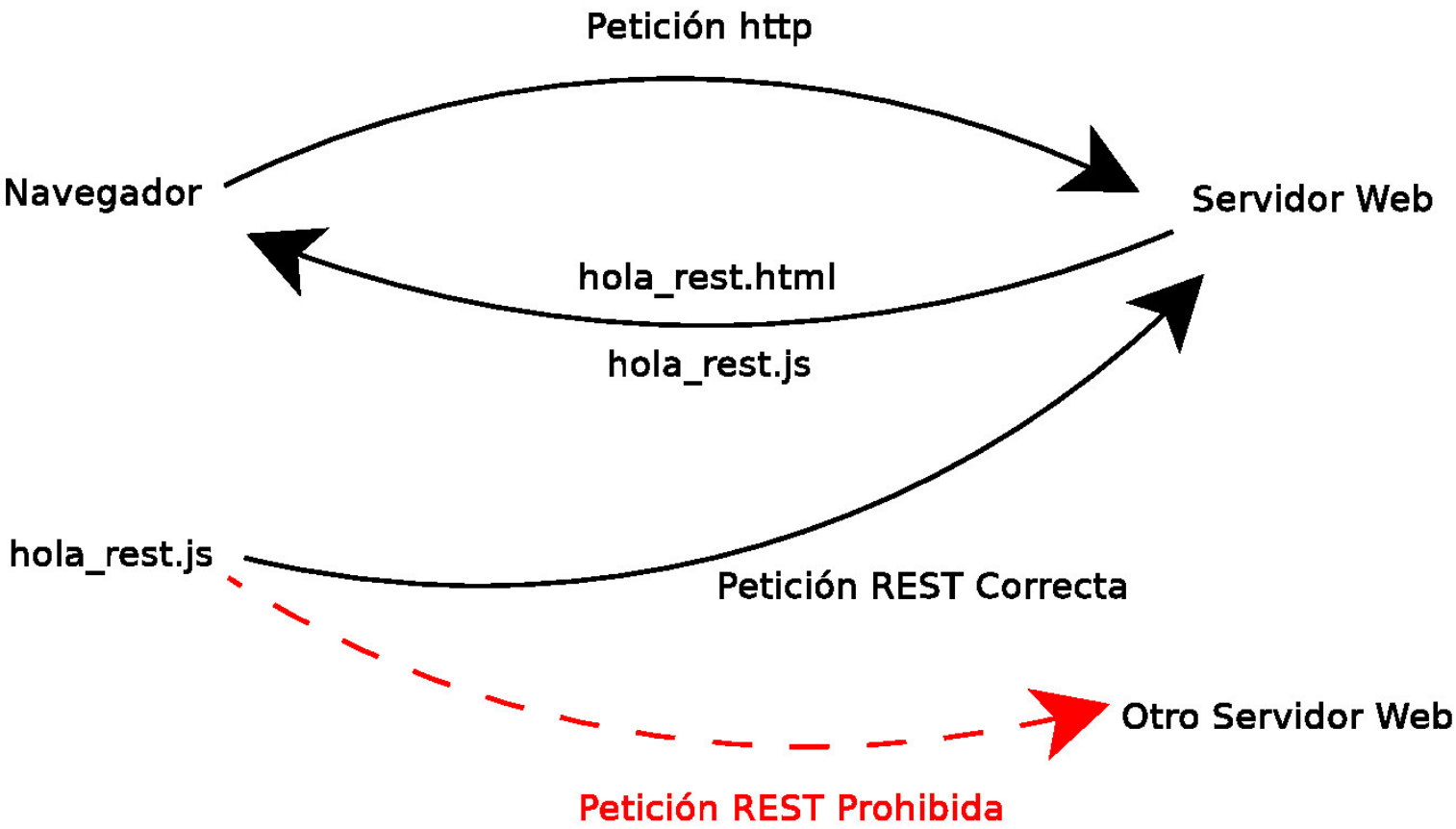


Figura: Same-Origin Policy

En ocasiones la *same-origin policy* resulta demasiado estricta y se requieren de técnicas que permitan, con el control adecuado, que un cliente haga peticiones a ciertos *orígenes* distintos

- JSONP (JSON with padding, JSON con relleno). Protocolo del año 2005. Obsoleto
- CORS. (Cross-Origin Resource Sharing). Protocolo desarrollado en 2004 y aceptado por el W3C en 2014

Aquí no lo veremos cómo aplicarlos, así que nuestros clientes REST estarán obligados a consultar con un servidor REST en el mismo origen (lo que en general es la opción preferible, la más sencilla y segura)

En la configuración que veremos aquí como ejemplo:

- Prepararemos un servidor con Express.js para que acepte peticiones REST/ROA en el puerto 3000 de *localhost*
- Por tanto, por la *same-origin policy*, el código JavaScript donde esté el cliente de este servidor, necesariamente debe haberse servidor por el servidor web en el puerto 3000 de *localhost*
- Por tanto, además de configurar Express.js para aceptar peticiones REST/ROA, también debemos configurarlo para servir los ficheros HTMl y JavaScript del cliente

En los ejemplos vistos en la asignatura hasta ahora, el navegador leía los ficheros HTML y los ficheros JavaScript directamente del sistema de ficheros local (el disco duro). Pero si siguiéramos trabajando así, incumpliríamos la *same-origin policy* (y no funcionaría a menos que configuráramos CORS)

Promesas

El uso de *promesas* es una técnica de programación concurrente disponible en lenguajes de programación como Java, JavaScript 6, C++, C#, Scala y muchos otros

- Una *promesa* es un objeto que inicialmente no tiene valor, pero que transcurrido algún tiempo, si todo va bien tendrá valor. O si no, lanzará una excepción
- En ECMAScript 2017 están disponible mediante las palabras reservadas *async await* (aunque hay otra sintaxis un poco más farragosas y antiguas)

Para extraer el valor de una promesa, anteponemos el operador *await*. Esto hace que la función se quede bloqueada esperando por el valor.

```
let respuesta =  await fetch(url);
```

El operador *await* solo se puede usar en funciones que hayan sido declaradas anteponiendo la palabra reservada *async*. Esto hace que la función sea asíncrona. También convierte el valor devuelvo por la función en una promesa

```
async function trae_resultado (numero) {
    [...]
    let respuesta =  await fetch(url);
    [...]
}
```

Aquí usaramos las promesas en la función *fetch()* que sirve para que un programa en JavaScript haga una petición HTTP, típicamente una petición REST/ROA

- A la función *fetch()* le pasamos una URL como argumento y devuelve una promesa con el resultado de la petición.
- Ese resultado es un objeto *Response* que contiene la respuesta HTTP en bruto, normalmente al programador lo que le interesará es el contenido en json. Para ello, el objeto *Response* dispone el método *json()* que devuelve una segunda promesa, con el objeto json.

```
async function trae_resultado (numero) {
  let dir_base = "http://localhost:3000/"

  let recurso = "api/dobla/";
  let url = dir_base + recurso + numero;
  console.log("url: " + url);

  let respuesta =  await fetch(url);
  console.log("respuesta: " + respuesta);

  let respuesta_json =  await respuesta.json();
  console.log("Respuesta_json: "+ respuesta_json);

  return respuesta_json;
}
```

- Si olvidamos el *await*, al trazar el valor, veremos que no contiene el json que esperaríamos, sino un objeto *promise*
- Si olvidamos añadir *async* a la función que usa *await*, obtendremos una excepción *await is only valid in async functions and the top level bodies of modules*

Recuerda: añadir *async* convierte el valor devuelto por la función en promesa. Por tanto, para leerlo, habrá que poner a otro *await*, que a su vez puede requerir otro *async*, que requerirá otro *await*... así hasta la función de más alto nivel del programa

- En otras palabras: cuando una función use una promesa, es necesario que esa función sea asincrona, y también su función *padre* (la función que la llama) y su abuelo, bisabuelo, tatarabuelo...

```
1   "use strict"
2  async function trae_resultado (numero) {
3    let dir_base = "http://localhost:3000/"
4
5    let recurso = "api/dobla/";
6    let url = dir_base + recurso + numero;
7
8    let respuesta =  await fetch(url);
9    let respuesta_json =  await respuesta.json();
10   return respuesta_json;
11
12 async function manej_boton01(event) {
13   //Número entero aleatorio entre 0 y 99
14   let numero = Math.floor(100 * Math.random());
15
16   span01.textContent = numero;
17   span02.textContent = await trae_resultado(numero);
18 }
19
20 let boton01 = document.querySelector("#boton01");
21 boton01.addEventListener("click", manej_boton01);
```

- Línea 8. Como *fetch()* devuelve una promesa, hay que anteponer *await*
- Línea 9. Como *json()* devuelve una promesa, hay que anteponer *await*
- Línea 2. Como en las líneas 8 y 9 hay un *await*, hay que anteponer *async*. Esto provoca que *trae_resultado* devuelva una promesa
- Línea 17. Como *trae_resultado* devuelve una promesa, hay que anteponer *await*
- Línea 12. Como en la línea 17 hay un *await*, hay que anteponer *async*

La función *manej_boton01* ya es de nivel global, no es necesario nigún *await* más (no se usa al registrar el manejador con *addEventListener* pero que esto no lee el valor devuelto por la función, solo lo vincula con el evento)

Ejemplo completo: doble de un número

En este ejemplo, tenemos un servidor REST/ROA que recibirá como parámetro un número y devolverá el doble de ese número

- Dirección base:
  `http://localhost:3000`
- Recurso:
  `/api/dobla/`
- Parámetro: `num`
- URL Completa:
  `http://localhost:3000/api/dobla/:num`

Ficheros requeridos

Necesitamos los siguientes ficheros:

- dobla_client.html
  Página web con el cliente
- dobla_client.js
  Código JavaScript del cliente
- dobla_server.js
  Script que configura Express para servir tanto las peticiones REST como los ficheros del cliente

```
http://localhost:3000/api/dobla/:num
http://localhost:3000/dobla_client.html
http://localhost:3000/js/dobla_client.js
```

Para probarlo en tu ordenador tendrás que:

1) Crear el directorio raiz para Express

   `~/www/site01/`

2) Copiar

   `dobla_client.html` a `~/www/site01/`

   `dobla_client.js` a `~/www/site01/js`

3) Lanzar Express.js con el fichero de configuración

   `node dobla_server.js`

4) Comprobar que el servidor atiende a las peticiones REST, pidiendo con tu navegador una página como p.e. `http://localhost:3000/api/dobla/10`
5) (opcional)
   Comprobar que Express sirve el código JavaScript, solicitando con el navegador la página
   `http://localhost:3000/js/dobla_client.js`
6) Comprobar que Express sirve la página web y que esta funciona correctamente
   `http://localhost:3000/dobla_client.html`

dobla_client.html

```html
<!DOCTYPE html>
<html lang="es-ES">

<head>
  <meta charset="utf-8">
  <title>Cómo enviar peticiones REST</title>
</head>

<body>
  <button id="boton01">Enviar un valor </button>
  <br>
  Valor enviado: <span id="span01"></span>
  <br>
  Valor recibido: <span id="span02"></span>

  <script src="js/dobla_client.js">
  </script>
</body>

</html>
```

http://ortuno.es/dobla_client.html

dobla_client.js

```
 "use strict"
async function trae_resultado (numero) {
  let dir_base = "http://localhost:3000/"

  let recurso = "api/dobla/";
  let url = dir_base + recurso + numero;
  console.log("url: " + url);

  let respuesta =  await fetch(url);
  console.log("respuesta: " + respuesta);
  let respuesta_json =  await respuesta.json();
  console.log("Respuesta_json: "+ respuesta_json);
  return respuesta_json;
}
```

```
async function manej_boton01(event) {
  //Número entero aleatorio entre 0 y 99
  let numero = Math.floor(100 * Math.random());

  span01.textContent = numero;
  span02.textContent = await trae_resultado(numero);
}

let boton01 = document.querySelector("#boton01");
boton01.addEventListener("click", manej_boton01);
```

http://ortuno.es/js/dobla_client.js.html

Atención en la construcción del path:

- Si lo generamos concatenando cadenas, hay que tener cuidado de que separando dirección base, recurso y argumentos haya exactamente una barra
  Es muy fácil cometer errores como

  `http://localhost:3000/api/dobla//:num`

  o

  `http://localhost:3000api/dobla/:num`

- Para evitarlos, en vez de concatenar las cadenas, podemos usar el método `path.join()` que se encarga de unir los fragmentos de URL, garantizado que haya una barra y solo una barra

```
path.join( dir_base, recurso, num );
```

dobla_server.js

```
const express = require('express');
const app = express();
const puerto = process.env.PORT || 3000;

app.use(express.json());
// middleware necesario para procesar las peticiones
// POST que incluyan un cuerpo json

// Direccionamiento estático:
const path = require('path');
// Importamos el módulo path

dir_raiz = path.join( process.env.HOME, "www/site01");
// Construye ~/www/site01

app.use(express.static(dir_raiz))
// Sirve todos los ficheros del directorio raiz
```

```
// Direccionamiento para GET
app.get('/api/dobla/:num', (req, res) => {
  let num = req.params.num;
  doblado = num * 2;
  // Normalmente aquí iría llamada a función en otro fichero

  console.log("El cliente envía " + num);
  console.log("La respuesta es " + doblado);
  doblado_json = JSON.stringify(doblado);
  res.json(doblado_json);
})
```

```
// Error 404
app.use((req, res) => {
  res.type('text/plain');
  res.status(404);
  res.send('404 - Dirección no encontrada');
})

// Error 500
app.use((err, req, res, next) => {
  console.error(err.message);
  res.type('text/plain');
  res.status(500);
  res.send('500 - Error en el servidor');
})

app.listen(puerto, () => console.log(
  `Express iniciado en http://localhost:${puerto}\n` +
  `Ctrl-C para finalizar.`));
```

http://ortuno.es/dobla_server.js.html

Captura del Status Code

Para leer el *status code* en el cliente, solo tenemos que leer la propiedad *status* del objeto promesa con la respuesta

- Si el código no es 200, significa que ha habido problemas y por tanto no podemos extraer ningún objeto json

```
async function trae_resultado (numero) {
  [...  obtiene la URL ...]

  let respuesta_bruto =  await fetch(url);
  let respuesta;  // Valor que devolverá esta función

  if (respuesta_bruto.status === 200) {
    respuesta = await respuesta_bruto.json();
    console.log("Respuesta correcta: "+ respuesta);
  else {
    respuesta = "Error " + respuesta_bruto.status;
    console.log(respuesta);}
  }

  return respuesta;
}
```