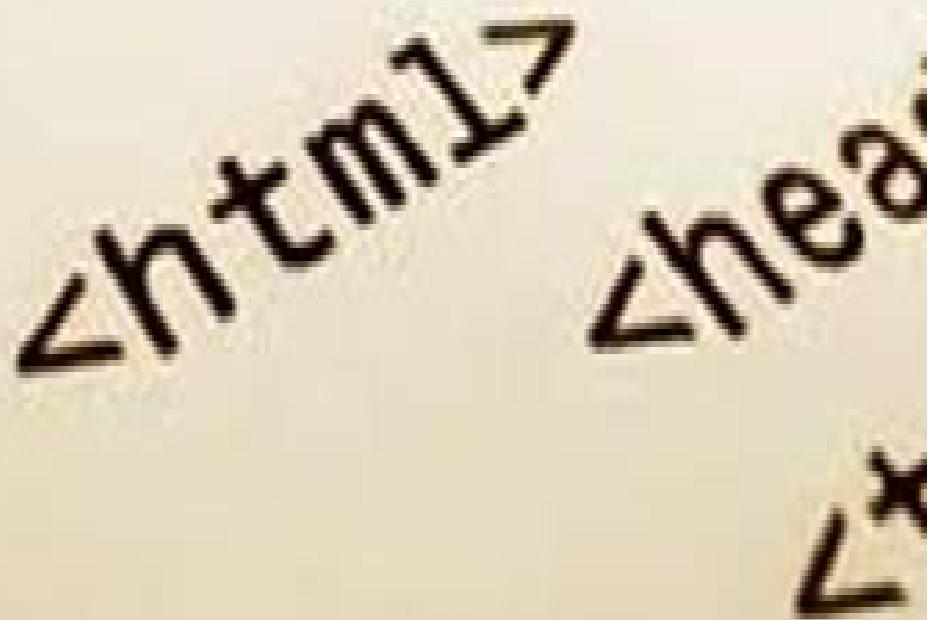




css



<html> <head> <*

CSS elemental

¿Será posible resumir todo lo que necesitamos saber sobre CSS en unas pocas páginas?

Realmente no pero, vamos a tratar de dar una vaga, vaga, muy vaga idea de lo que algunos consideran un “lenguaje” complicado pero, que no es un lenguaje y en el fondo es algo bastante simple.

CSS es una tecnología desarrollada por el W3C (World Wide Web Consortium) con el fin de separar el contenido de la presentación en una página web; es decir, agrupar las definiciones de las etiquetas lo que nos permite hacer que las páginas sean más livianas ya que si no, deberíamos agregarles esas propiedades a cada etiqueta, repetirlas una y otra vez cada vez que las quisiéramos usar. De esta manera, definimos el estilo de un sitio completo de una sola vez y luego, si es necesario, agregamos estilos individuales.

Índice de contenido

Introducción	5
La sintaxis básica	7
Los selectores	9
Jerarquía y herencia	12
Las unidades	16
Colores y paletas	19
Gradientes	22
background: lo standard y lo nuevo	25
border y outline: los recuadros	33
margin y padding	37
Alineación	41
Dimensiones y visualización	43
Posición	47
Cuando las cosas se desbordan	51
Las fuentes de los textos	54
Los textos	58
Los cursores	64
Propiedades de las listas	65
Propiedades de las tablas	67
Transformaciones	69
Filtros	72
Transiciones	78
Animaciones	81
Columnas	83
Flotaciones	87
Selectores especiales	96
Selectores de atributos	98
Pseudo-clases	100
Pseudo-elementos	108
Navegadores y prefijos	113
at-rule	115
Variables y propiedades personalizadas	118
Cajas flexibles	120
Referencias	126
Dudas y errores cuando se usa CSS	127
El ID siempre debe ser único	132
Sobre IDs y clases	133
Las propiedades CSS por defecto	136
Un espacio hace la diferencia	138
Algo sobre colores	140
Un poco más allá de las imágenes	144
Múltiples fondos con CSS3	147
Los misterios de background-size	149
Los misterios de background-clip	152
Sobre el uso de los sprites	154
Los bordes ocupan espacio	156
margin versus padding	157

<u>Alinear verticalmente</u>	163
<u>Sobre IDs y bloques ocultos</u>	165
<u>Las posiciones absolutas son relativas a “algo”</u>	168
<u>Los pixeles son unidades indivisibles</u>	170
<u>Colocar texto alrededor de una imagen</u>	171
<u>Ajustar el tamaño de una imagen al texto</u>	173
<u>Flotaciones, fondos, problemas, alternativas</u>	176
<u>Optimizar el CSS</u>	179
<u>Consejos para trabajar con CSS</u>	183
<u>El selector universal</u>	185
<u>Identificar enlaces por sus atributos</u>	186
<u>Ejemplos de nth-child</u>	187
<u>Eventos click y CSS</u>	189

Introducción

Escribamos una página web sencilla:

```
<!DOCTYPE html>
<html>
  <head>
    <title>página demo</title>
    <meta charset="utf-8">
    <meta content="text/html; charset=UTF-8" http-equiv="Content-Type">
  </head>
  <body>
    <h1>título principal</h1>
    <div>
      <p>este es un texto cualquiera</p>
      <p><strong>y este es otro texto</strong></p>
    </div>
    <h2>subtítulo</h2>
    <p><em>con un texto en itálica</em></p>
    <div>
      <p>este es otro texto cualquiera con <a href="#">un enlace</a></p>
      <p>y otro texto más</p>
    </div>
  </body>
</html>
```

Guardemos eso en un archivo con extensión .html y miremos el resultado en cualquier navegador:

título principal

este es un texto cualquiera

y este es otro texto

subtítulo

con un texto en itálica

este es otro texto cualquiera con un enlace

y otro texto más

Lógicamente, esta es una página muy sencilla así que difícilmente pueda deslumbrar a alguien pero ¿no podemos tratar que se vea un poco más adornada?

Quisiera que se viera centrada, con otro color de fondo, con otro tipo de letras, con algún recuadro y además, odio los enlaces azules y subrayados.

Todas esas cosas y todas las otras que se nos puedan ocurrir es justamente lo que da sentido a la existencia del CSS.

Unas cuantas reglas de estilo y ahora se vería algo así:

título principal

este es un texto cualquiera
y este es otro texto

subtítulo

con un texto en itálica

este es otro texto cualquiera con un enlace
y otro texto más

Tampoco es algo maravilloso pero vamos mejorando.

Sin CSS, una página está desnuda pero, aún así, algo de pudor le queda ya que los navegadores establecen una serie de propiedades por defecto que serán aplicadas a menos que las cambiemos. Tienen ciertas diferencias según se trate de un navegador u otro pero, en general, son justamente esas propiedades no identificadas, las que nos causan problemas porque no solemos tenerlas en cuenta.

[VER REFERENCIAS \[Las propiedades CSS por defecto\]](#)

La sintaxis básica

CSS (Cascading Style Sheets) es un mecanismo para adicionar un estilo (fuentes, colores, etc) a un documento HTML.

No es un lenguaje de programación sino un “listado” de propiedades organizado bajo ciertas reglas de sintaxis.

Por ejemplo, para establecer que el texto incluido en una etiqueta `<h1>` se muestre de color azul escribiríamos esto:

```
h1 {color: blue;}
```

Esto es lo que se llama una regla y consiste en dos partes, la primera se denomina selector (`h1`), la segunda declaración (`color:blue`) que, a su vez tiene dos partes, la propiedad (`color`) y el valor (`blue`).

El selector es un vínculo entre el HTML y la hoja de estilo. Todos los elementos (etiquetas) de un documento son selectores y cada uno de ellos posee un conjunto de propiedades generales y particulares.

Entonces cualquier regla de CSS debe escribirse entre llaves y en su interior se enumeran las propiedades, separándolas con un punto y coma:

```
selector {  
    propiedad_1: valor;  
    propiedad_2: valor;  
}
```

Hay tres formas de establecer reglas de estilo. La primera es incluirlas dentro de la etiqueta `<style>`

```
<style>  
    h1 {color: blue;} /* esto es un comentario */  
    h2 {color: red;}  
</style>
```

La segunda es guardándolas en una hoja de estilo que insertamos en el `<head>` de nuestra página:

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

¿Y qué es una hoja de estilo? Un archivo de texto plano sin formato (generalmente de extensión `.css`), que podemos crear con el *block* de notas y que NO INCLUYE etiquetas y no es otra cosa que el contenido de eso que ponemos en una etiqueta `<style>`

De alguna manera, podemos decir que ambos métodos pueden combinarse ya que dentro de una etiqueta `<style>` podemos “importar” una hoja de estilo externa:

```
<style>
  @import url(archivo.css);
  h1 {color: blue;}
</style>
```

El tercer método es mediante el atributo **style** de cualquier etiqueta:

```
<p style="color: green">
  este texto se vera con letras verdes
</p>
```

Para reducir el tamaño de las hojas de estilo, podemos agrupar los selectores, separándolos mediante comas. Por ejemplo, de este modo podríamos establecer la misma regla para tres etiquetas distintas:

```
h1, h2, h3 {color: blue;}
```

Como ya vimos, también pueden agruparse las declaraciones separándolas con un punto y coma o repetir selectores:

```
h1, h2, h3 {color: blue;}
h1 {font-size: 24px;}
```

Como muchas veces, las reglas de estilo son engorrosas, conviene ir escribiendo qué hacen o para qué las usamos.

En CSS, los comentarios se indican colocando el texto entre /* y */. Una muestra de comentario tendría el siguiente formato:

```
p {color: aqua;} /* este se el color de los párrafos */
```

Ya veremos que algunas propiedades pueden ser definidas de dos maneras distintas, una resumida donde se establecen varias de modo simultáneo separándolas con espacios y otra sencilla donde el valor es único.

```
p {background: red url() no-repeat 0 0;}
p {background-color: red;}
```

[VER REFERENCIAS \[Consejos para trabajar con CSS\]](#)

Los selectores

El selector es el elemento que está conectado a un estilo particular. Como dijimos, cualquier etiqueta HTML es un posible selector CSS.

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

Pero está claro que eso es poco útil ya que las etiquetas que usamos suelen ser pocas pero muy repetidas y necesitamos referenciarlas de otro modo.

Para permitir este control es que las etiquetas poseen dos atributos con los cuales podemos identificarlas. El atributo **id** se utiliza para dar un nombre a una etiqueta que queremos que tenga un estilo específico. Un nombre que debe ser único, sólo debe haber uno en toda la página y debería reservarse para usar con elementos importantes.

```
<h1>título</h1>

<div id="principal">
  <p>un texto</p>
  <p class="ejemplo">otro texto</p>
  <em>texto en itálica</em>
</div>

<h2 class="ejemplo">subtítulo</h2>
<p><em>texto en itálica</em></p>

<div>
  <p>un texto</p>
  <p class="ejemplo">otro texto</p>
</div>
```

Ahora, podemos definir reglas de estilo para ese elemento anteponiéndole el carácter #:

```
div {color: blue;}
#principal {color: red;}
```

Los textos del **<div>** al que llamamos **#principal** serán rojos y cualquier otro será azul.

VER REFERENCIAS [El ID siempre debe ser único]

El otro atributo que permite esto es **class** y es posible utilizarlo en múltiples etiquetas y de ese modo agrupar estilos repetitivos. Las clases también son nombres que nosotros creamos y se referencian anteponiendo un punto:

A lo anterior agregamos:

```
.ejemplo {color: green;}
```

Ahora, los párrafos con esa clase serán de color verde, sin importar dónde están e incluso, sin importar si son párrafos; cualquier etiqueta con esa clase se verá verde, incluyendo la etiqueta `<h2>`

Si quisieramos ese color sólo en la etiqueta `<p>` deberíamos cambiar la regla:

```
p.ejemplo {color: green;}
```

Que si la leemos dice algo así como: toda etiqueta `<p>` cuya atributo `class` sea `ejemplo` será de color verde.

Así que, el atributo `class` nos permite crear estilos que se pueden utilizar cuando queramos. Primero definimos una regla en el documento CSS (con el punto inicial obligatorio) y luego podemos aplicar la clase a cualquier etiqueta HTML.

Demás una etiqueta puede tener diferentes clases permitiendo así que el mismo elemento pueda tener diferentes estilos.

```
p.ejemplo1 {color: red;}  
p.ejemplo2 {color: green;}
```

Los selectores de contexto son cadenas de dos o más selectores simples separados por un espacio en blanco.

```
p em {color: yellow;}
```

Esta regla dice que el texto con énfasis (``) dentro de un párrafo (`<p>`) deberá ser de color amarillo. De ese modo, el texto con énfasis dentro de cualquier otra etiqueta no sería afectado.

VER REFERENCIAS [Dudas y errores cuando se usa CSS]

Por lo general asignamos una clase CSS a una etiqueta pero, no hay nada que nos impida asignarle varias simultáneamente. Por ejemplo:

```
<p class="claseUno"> ... contenido ... </p>  
<p class="claseUno claseDos"> ... contenido ... </p>
```

Cuando utilizamos varias clases juntas, simplemente las sepáramos con un espacio. Pero ¿y si ambas clases tienen propiedades similares?

Supongamos que tenemos una clase que define al color como rojo y otra como verde.

```
.claseColorVerde {color: green;}  
.claseColorRojo {color: red;}
```

Si la primera clase es `claseColorRojo` veremos todo en rojo:

```
<p class="claseColorRojo claseColorVerde"> ... contenido ... </p>
```

Y si la primera clase es *claseColorVerde* ... TAMBIÉN VEREMOS TODO EN ROJO. ¿No hay diferencias? ¿Por qué no se ve de color verde?

```
<p class="claseColorVerde claseColorRojo"> ... contenido ... </p>
```

Porque en este caso, el orden en que las coloquemos en el atributo es indiferente.

El HTML es secuencial, se ejecuta de arriba hacia abajo en el orden en que esté escrito y, como primero hemos definido la regla verde y luego la regla roja, esta última sobrescribe la anterior y por lo tanto, la primera no se ejecuta.

En resumen, si hay propiedades “repetidas”, siempre se le dará prioridad a la última que hayamos definido.

VER REFERENCIAS [Sobre IDs y clases]

Hay un selector especial que es un simple asterisco y es el llamado selector universal que suele ser la primera regla que se agrega a cualquier hoja de estilo porque de esa manera se pueden resetear las propiedades por defecto y nos permiten definirlas a todas.

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

VER REFERENCIAS [El selector universal]

¿Demasiado? Respirar hondo y seguir adelante.

Jerarquía y herencia

Para poder utilizar CSS y sacarle el máximo provecho es indispensable entender cuál es la estructura de un documento HTML.

Podemos imaginarnos que una página web es un conjunto de cajas (bloques) metidas una dentro de la otra, siendo la más grande, la que contiene a todas, el elemento definido con la etiqueta `<html>`.

En este tipo de estructura, donde hay elementos contenidos en otros hay una cierta jerarquía, los elementos padre son los contenedores y los elementos hijos los contenidos. El problema de eso es que como las etiquetas pueden anidarse, hay que descubrir cuál está dentro de cuál para conocer el orden de prioridades y eso es lo que se llama herencia.

Muchas propiedades de estilo de los elementos padre son heredadas por los elementos hijo, pero no al revés.

Supongamos que tenemos este texto con cuatro animales para que sea más claro:

```
<p>león <strong>jirafa<em> elefante</em></strong> <em>perro</em></p>
```

Sin reglas de estilo definidas, veríamos algo así:

león jirafa *elefante* perro

Todo tendrá la misma fuente, “jirafa” se mostrará en negrita (etiqueta ``), “elefante” se mostrará en itálica (etiqueta ``) pero también en negrita porque `` está dentro de ``.

Decimos entonces que `` desciende de `` que es su etiqueta “padre” (*parent*) y a su vez, todas descienden de `<p>` que sería el “abuelo”.

Si establecemos una propiedad de color para la etiqueta `<p>`, las demás heredarán esa propiedad; por ejemplo, así, todo el texto será rojo:

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

En cambio, si agregamos esta regla:

```
strong {color: green;}
```

Tanto *jirafa* como *elefante* serán de color verde y el resto será rojo.

Por último, si agregamos:

```
strong em {color: black;}
```

Veremos esto:

león jirafa elefante perro

Y *perro* no cambiará de color porque la regla dice que sólo será negro el contenido de una etiqueta `` que esté DENTRO de ``. Todos los selectores que estén anidados dentro de otros heredarán los valores de propiedades asignados al selector exterior, salvo que se modifiquen de otra manera.

Este tipo de selector se denomina contextual y consisten en una serie de elementos separados por un espacio. Esto puede tener múltiples opciones y combinaciones:

```
ul li {font-size: small;}
ul ul li {font-size: x-small;}
div p {font: small sans-serif;}
.ejemplo1 h1 {color: red;}
#ejemplo2 code {background: blue;}
div.ejemplo3 h1 {font-size: large;}
h1 b, h2 b, h1 em, h2 em {color: red;}
```

Cuando agregamos reglas o propiedades CSS no sólo debemos cuidar que la sintaxis sea correcta. Muchas veces, aunque todo parezca perfecto, nos vamos a encontrar con que ... no parecen funcionar. Por lo general, esto tiene una explicación: en el CSS hay un orden de prioridades que a veces tiene que ver con el orden real (qué está antes y qué está después) pero también con el valor específico del selector que usamos.

Un ejemplo, supongamos que tenemos este código HTML:

```
<p class="azul" id="rojo">un texto</p>
```

Y estas definiciones de estilo:

```
#rojo {color: red;}
.azul {color: blue;}
```

A la etiqueta le estamos agregando dos atributos, `id` y `class`, en uno le decimos que el párrafo debe ser azul y en el otro que debe ser rojo ¿De qué color se verá el texto?

Ya vimos que si usamos dos clases con las mismas propiedades, el orden importa pero en este caso no es así ya que sea como sea que lo escribamos, siempre será rojo porque un `id` es “más importante” que una clase.

Los selectores tienen un orden de prioridades, algo que la w3.org dice que podemos calcular con aritmética pero, por lo general, basta saber que esas prioridades existen para aprovecharse de ellas; el orden es más o menos este:

1. lo que está definido en la etiqueta en el atributo `style`
2. lo que está definido con un atributo `id`
3. lo que está definido con un atributo `class`
4. lo que está definido con un selector genérico (una etiqueta)

Por ejemplo:

```
<style>
  p {color: red;}
  p.verde {color: green;}
</style>

<p>esto se verá rojo porque TODOS los párrafos son rojos</p>
<p class="verde">pero esto se verá verde</p>
<p style="color: yellow;">este texto se verá de color amarillo</p>
```

De esta manera sobreescrivimos la regla. Esto es lo que se llama jerarquía. Si hay varias reglas diferentes que afectan a un mismo elemento, sólo se aplicará una de ellas.

Esto será así incluso si ponemos todo junto:

```
<p class="verde" style="color: yellow;">este texto se verá de color amarillo</p>
```

Lo mismo ocurre con definiciones más complejas como las listas; por ejemplo, si queremos resaltar un ítem:

```
<ul id="demo">
  <li class="resaltado">primer ítem de una lista</li>
  <li>segundo ítem de una lista</li>
</ul>
```

Definirlo así, no funcionará:

```
ul#demo li {color: blue;}
.resaltado {color: red;}
```

Porque lo que dice `ul#demo li` es más “poderoso” que lo que dice la clase `.resaltado`; ahí es donde funciona bien la aritmética para calcular esas prioridades o el instinto que nos dice que podríamos hacer esto:

```
ul#demo li {color: blue;}
ul#demo li.resaltado {color: red;}
```

La herencia y la jerarquía tiene una excepción con la cual podemos “forzar” las reglas: la posibilidad de usar la palabra `!important` que podemos agregar al final de una propiedad CSS y que realza cierto valor para, de esta forma, obligar a que se ignore una declaración y forzar un cambio.

Habíamos visto un ejemplo donde usábamos dos clases y el orden en que las escribíamos definía cuál era la que se iba a ver; en este caso era rojo porque era la última:

```
.claseColorVerde {color: green;}
.claseColorRojo {color: red;}

<p class="claseColorVerde claseColorRojo"> ... contenido ... </p>
```

Sin embargo, podríamos cambiar la primera “forzando” a que fuera verde:

```
.claseColorVerde {color: green !important;}
```

Otro ejemplo:

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

Los enlace serán azules pero si ponemos esto, los enlace serán rojos:

```
a {color: red !important;}  
a {color: blue;}
```

Entonces, si bien la definiciones CSS se ejecutan de manera secuencial y la última sobrescribe a la anterior, hay una excepción y es cuando se encuentra la palabra **! important**. En ese caso, todas las demás declaraciones son ignoradas y por ese motivo debemos ser prudentes con el uso de esta palabra y sólo aplicarla cuando es absolutamente necesario ya que si no, nos veremos en problemas a la hora de corregir algo.

Muchas propiedades son heredadas de alguna otra (hasta llegar a la raíz de la página) y hay una forma en que podemos indicar eso de modo explícito: la palabra **inherit**:

```
p {color: inherit;}
```

Las unidades

Como ya vimos, toda propiedad consta de dos partes, su nombre y su valor separados por dos puntos:

propiedad : valor;

El valor es la asignación que recibe y depende del tipo de propiedad. Puede ser una longitud, una dirección, un color, una función o una palabra específica.

Las unidades de longitud nos permiten definir el tamaño de los elementos y de los textos y hay dos grandes grupos: las unidades absolutas y las unidades relativas.

Las unidades absolutas sólo dependen del medio de salida (por ejemplo, una impresora) así que son muy dependientes de este y se dice que sólo son útiles si se conoce cuál es ese medio:

cm	centímetros
in	pulgadas (1 pulgada = 2.54 centímetros)
mm	milímetros
pc	picas (1 pica = 12 puntos)
pt	puntos (1 punto = 1/72 pulgada)
px	pixeles (relativa a la resolución de la pantalla)
q	un cuarto de milímetro

Las unidades llamadas relativas son siempre relativas a algo, es decir, se miden respecto a otra cosa. Especifican una longitud relativa a otra propiedad de longitud ya definida y suelen ajustarse mejor a dispositivos diferentes:

%	porcentajes
ch	relativa al ancho del carácter "0"
em	relativa a la altura de la fuente
ex	relativa a la altura de la letra "x"
rem	se refiere al tamaño de la fuente del elemento <html>

Las posibilidades son varias aunque, por lo general, sólo usamos tres: pixeles (**px**), relativas (**em**) o porcentajes (%).

¿Es mejor una que otra? Las opiniones varían así que no hay que hacer caso a los consejos y usar las que nos resulten más cómodas.

En la mayoría de los casos, usar pixeles (**px**) proporciona un control más efectivo que el resto de las unidades absolutas ya que los navegadores los interpretan correctamente.

Entre las unidades relativas los más utilizados son los porcentajes y **em** que hay que recordar que siempre dependen del valor de un elemento anterior que, pudo ser establecido por nosotros o no.

```
font-size:1em; /* este es el tamaño por defecto */  
font-size:1.5em; /* un valor mayor, significa un tamaño superior */  
font-size:0.5em; /* un valor menor, significa un tamaño inferior */
```

Si bien no es sencillo de controlar, usar **em** permite crear páginas proporcionadas que no se deformarán sin importar donde las veamos. De alguna manera son similares a la unidad porcentaje pero los primeros se suelen usar para las fuentes de los textos y los segundos para las dimensiones, generalmente, con las propiedades **width** y **height**.

Digo generalmente ya que nada impide definir dimensiones con cualquier tipo de unidad:

```
width:10em /* es correcto */
```

Cuando usamos unidades de longitud es importante recordar que no debe quedar un espacio entre el número y esa unidad:

```
width:10 em /* es incorrecto */  
width:10em /* es correcto */
```

VER REFERENCIAS [Un espacio hace la diferencia]

La unidad de longitud debe existir siempre salvo que el valor sea cero en cuyo caso es optativa. Esto es correcto:

```
margin: 0;
```

Dependiendo de cada propiedad, las longitudes pueden tener valores positivos o negativos.

```
margin-left: 10px;  
margin-left: -20px;
```

VER REFERENCIAS [Los pixeles son unidades indivisibles]

Hay una serie de unidades nuevas que permiten establecer valores relativos al tamaño de la pantalla (o ventana) del dispositivo:

vh	equivale a 1/100 de su alto
vw	equivale a 1/100 de su ancho
vmin	equivale a 1/100 de la relación mínima entre el alto y el ancho
vmax	equivale a 1/100 de la relación máxima entre el alto y el ancho

Hay propiedades que requieren valores expresados como ángulos y en ese caso, se puede usar cualquiera de estas:

deg	el ángulo se expresa en grados
grad	el ángulo se expresa en grados centesimales
rad	el ángulo se expresa en radianes
turn	el ángulo se expresa en giros

La unidad más utilizada es **deg** y basta tener en cuenta que, a diferencia de las unidades de longitud, no se admite el uso de un valor de cero sin que se especifique la unidad:

$$0\text{deg} = 0\text{grad} = 0\text{turn} = 0\text{rad}$$

Otro tipo de valor usual es el tiempo que se debe expresar en milisegundos (**ms**) o segundos (**s**).

Al igual que con los ángulos, siempre debe agregarse la unidad aunque el valor sea cero.

Otro tipo de valores a usar son los nombres de los colores (*aqua*, *red*, *black*), su código hexadecimal (#ffffff, #ffff00, #bf0044) o palabras pre-establecidas como (*left*, *solid*, *none*, *visible*, etc).

La función **calc()** puede usarse en cualquier propiedad que utilice como valor una longitud, ángulo o cualquier tipo de “número” y lo que hace es “calcular” ese valor utilizando una expresión aritmética que puede incluir sumas (+), restas (-), multiplicaciones (*) o divisiones (/).

Debe tenerse en cuenta que los operadores de sumas (+) y restas (-) siempre deben estar separados por un carácter espacio; por el contrario, para el resto de los operadores esto no es necesario aunque se recomienda hacerlo de todos modos.

Por ejemplo:

width: calc(100% - 80px);

calc() es útil para posicionar elementos o dimensionarlos cuando el espacio disponible es variable.

```
p {  
    width: calc(100% - 1em);  
}
```

Colores y paletas

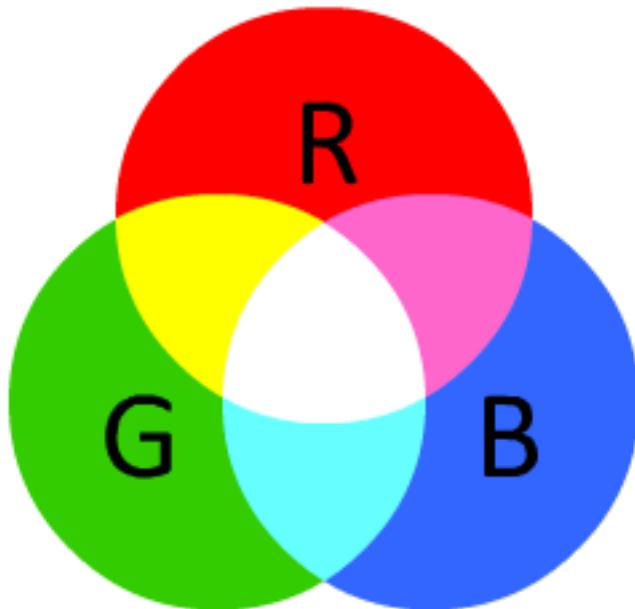
El valor de los colores se representa de varias formas. La más utilizada es mediante la combinación de tres valores (RGB *red green blue*) entre 0 y 255, esto significa que una paleta de colores puede tener hasta $256^3 = 16777216$ colores diferentes.

Sin embargo, tanto los sistemas operativos como los programas tienen limitaciones para mostrarlos adecuadamente por lo que “intentan simularlos” representando el color más cercano, además, cada usuario personaliza su monitor con distintos grados de brillo y saturación y por eso, muchos visitantes verán colores “diferentes” a los que creemos que colocamos.

En la paleta de colores seguros se agrupan aquellos que la mayoría puede mostrar sin distorsiones y consta de 216 colores, donde sólo se utilizan 6 valores hexadecimales (00, 33, 66, 99, CC y FF)

¿Por qué no son 256?, porque los sistemas operativos reservan algunos de esos colores básicos para su uso interno, los llamados colores del sistema.

En HTML y CSS, los colores también pueden definirse por un nombre, los 16 colores elementales son: *aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white* y *yellow*.



El formato RGB puede ser indicado como valor hexadecimal anteponiendo el carácter # pero también puede ser expresado mediante una función llamada `rgb()` donde los tres argumentos a incluir deben estar separados por comas y pueden ser tanto números en el rango 0-255 como porcentajes.

Entonces, los colores RGB se pueden indicar de cuatro maneras:

#rrggbb por ejemplo #00CC00

#rgb por ejemplo #OC0

rgb(x,y,z) mediante enteros entre 0 y 255 inclusive

rgb(x%,y%,z%) mediante números entre 0.0 y 100.0 inclusive

Los códigos de color los utilizamos para establecer el color de fondo de los elementos, los bordes, las fuentes, etc:

color: #FF0000;

color: rgb(255,0,0);

color: rgb(100%,0%,0%);

color: red;

Todos esos ejemplos harán lo mismo, mostrar algo de color rojo. Eso es lo que se llama el modelo RGB donde cada color está compuesto por una parte de rojo (*red*), una parte de verde (*green*) y una parte de azul (*blue*) y las diferentes proporciones de estos componentes generan todos los colores posibles que, la aritmética nos dice que son:

$256^*256^*256^* = 256^3 = 16777216$ colores diferentes

A medida que la tecnología avanzó, la cantidad de colores disponibles fue aumentando. En algún momento pasamos del “blanco y negro” a soportar 8 colores, luego 16 (4 bits), luego 256 (8 bits), luego 65536 (16 bits) y ahora el llamado *TrueColor* (24 bits).

Cualquiera diría que 16 millones de colores es más que suficiente ya que el ojo humano difícilmente distingue ciertas variaciones sutiles pero, ahora nos enfrentamos con una nueva alternativa, los llamados colores RGBA (*red green blue alpha*) que agregan un nuevo factor, el *alpha channel*, es decir, la opacidad o transparencia que sigue el mismo esquema de porcentajes: el 0% representa la transparencia absoluta y el 100% representa la opacidad absoluta que es la forma en que tradicionalmente vemos los colores.

Esto es fácilmente visible cuando vemos algunas imágenes en formato PNG y no es un avance sin costo ya que pasamos de 24 bits a 32 bits es decir, imágenes más pesadas.

Lo interesante es que el uso de este canal *alpha* no está limitado a las imágenes, también es posible aplicarlo a algunas propiedades CSS.

El valor es expresado como porcentaje decimal, yendo de 0 (transparencia total) hasta 1 (el valor por defecto). En este ejemplo, al color rojo se le aplicaron diferentes valores al canal *alpha*:



Otro formato que puede usarse para establecer el valor de un color son los llamados **hsl()** (*hue saturation lightness*) y **hsla()** (*hue saturation lightness alpha*).

Este sería el color rojo normal o con una transparencia del 50%:

```
color: hsl(0,100%, 50%);  
color: hsla(0,100%, 50%, 50%);
```

[VER REFERENCIAS \[Algo sobre colores\]](#)

Gradientes

Otra de las nuevas características del CSS3 es la que nos permite crear gradientes de color para utilizar como fondos. Para esto hay dos propiedades: **linear-gradient** y **radial-gradient** y las utilizamos como si fueran un color dentro de la propiedad **background**:

linear-gradient(inicio angulo, color1, color2);

radial-gradient(inicio angulo, shape size, color1, color2);

Donde:

inicio es la posición desde donde comenzará a ser generada; pueden ser valores expresados en pixeles, porcentajes o palabras clave como **left**, **top**, **right**, **bottom** o **center**.

angulo obviamente, indica el ángulo del gradiente

shape puede ser **circle** o **ellipse** y define la forma de una gradiente

size indica la forma en que se expandirá y terminará y puede ser **closest-side** (o **contain**), **closest-corner**, **farthest-side**, **farthest-corner** (o **cover**)

colorX indica los colores que se utilizarán y como mínimo colocaremos dos aunque puede haber más

Algunos ejemplos degradados lineales y radiales:

background: linear-gradient(#123, #789);

background: linear-gradient(#123, #789, #BBO);

background: radial-gradient(#123, #789);

background: radial-gradient(#123, #789, #BBO);

Quizás es algo confuso pero, lo básico es entender que una gradiente no es otra cosa que un degradado de colores que se aplica a un fondo así que, lo más simple es crear una de dos colores y eso lo indicamos colocando como parámetros, esos dos colores, en formato hexadecimal, **rgb()**, **rgba()**, **hsl()** o **hsla()**:

background: linear-gradient(#FFF, #000);

Así, sin más datos, creamos una gradiente que va del blanco al negro y que llenará el espacio al que lo aplicamos, de modo proporcional.

Tal como está, el degradado será vertical pero la dirección es algo que podemos definir. La posición, indica dónde comenzará la gradiente y requiere de dos palabras o valores al igual que la propiedad **background-position** (**top**, **right**, **bottom**, **left**, **center**); si sólo ponemos una, se asume que la otra es **center** así que estos cinco colores se distribuyen parejos.

background: linear-gradient(0deg, #000000, #FF0000, #00FF00, #0000FF, #FFFFFF);

Y en este ejemplo, el verde (#00FF00) comienza 100 pixeles a la izquierda:

```
background: linear-gradient(0deg, #000000, #FF0000, #00FF00 100px, #0000FF, #FFFFFF);
```

Más ejemplos:

```
background: linear-gradient(center top, #FFF, #000);  
background: linear-gradient(top, #FFF, #000);
```

```
background: linear-gradient(center bottom, #FFF, #000);  
background: linear-gradient(bottom, #FFF, #000);
```

```
background: linear-gradient(left center, #FFF, #000);  
background: linear-gradient(left, #FFF, #000);
```

```
background: linear-gradient(right center, #FFF, #000);  
background: linear-gradient(right, #FFF, #000);
```

Si queremos que los colores inviertan su orden reemplazamos **top** por **bottom**; si queremos que sea horizontal (de izquierda a derecha) usamos **left** y viceversa usamos **right**.

Ahora, agregaremos un valor más, el ángulo que debe ser expresado en grados lo que permite establecer la dirección del gradiente:

```
background: linear-gradient(0deg, #FF0, #OFF);  
background: linear-gradient(90deg, #FF0, #OFF);  
background: linear-gradient(180deg, #FF0, #OFF);  
background: linear-gradient(270deg, #FF0, #OFF);
```

También podemos combinar varios tipos de gradientes, separándolas con comas:

```
background: linear-gradient(#000,#ABC,transparent), linear-gradient(right,#880, #234);
```

Bastaría seguir agregando colores separados por comas para que la gradiente se hiciera cada vez más complicada:

```
background: linear-gradient(#FFF, #FF0, #000);  
background: linear-gradient(#FFF, #F00, #FF0, #000);  
background: linear-gradient(#FFF, #F00, #FF0, #0F0, #000);
```

Los colores se muestran ordenados y se distribuyen proporcionalmente pero, basta agregarles un valor separado por un espacio para indicar la posición de cada uno:

```
background: linear-gradient(#FFF, #FF0 20%, #000);
```

Hasta acá, la gradiente la aplicamos a la propiedad **background** pero podríamos colocarla en la propiedad **background-image** y, de ese modo agregar colores de fondo, algo que tiene sentido si se utilizan colores en forma **rgba()** que nos dan la posibilidad de crear fondos con determinada opacidad:

```
background-color: red;  
background-image: linear-gradient(#000, rgba(200,150,200,.5), #000);
```

Vemos que así como hay gradientes lineales, también las hay de tipo radial donde el degradado se genera de modo circular.

Si nos limitamos a lo simple, la sintaxis de ambas no difiere demasiado:

```
background: radial-gradient(#FFF, #000);
```

Como en las lineales, la posición indica dónde comenzará la gradiente (**top**, **right**, **bottom**, **left**, **center**):

```
radial-gradient(center top, #FFF, #000);  
background: radial-gradient(top, #FFF, #000);
```

```
background: radial-gradient(center bottom, #FFF, #000);  
background: radial-gradient(bottom, #FFF, #000);
```

```
background: radial-gradient(left center, #FFF, #000);  
background: radial-gradient(left, #FFF, #000);
```

```
background: radial-gradient(right center, #FFF, #000);  
background: radial-gradient(right, #FFF, #000);
```

Claro que esto se complica un poco más ya que este tipo de gradientes posee otro parámetro opcional que podemos utilizar y que consta de dos palabras **shape** (la forma) y **size** (el tamaño).

```
background: radial-gradient(circle closest-side, #FFF, #000);
```

```
background: radial-gradient(left top,circle closest-side, #FFF, #000);
```

La propiedad **repeating-linear-gradient()** crea una imagen con gradientes repetidas y se utiliza con los mismos parámetros que la gradiente normal. Sólo se diferencia en que el color final coincide con el primer color y si son distintos, se produce una transición. Por ejemplo:

```
background-image: repeating-linear-gradient(45deg, yellow, red);
```

La propiedad **repeating-radial-gradient()** es similar pero se ejecuta en gradientes radiales-

```
background-image: repeating-radial-gradient(red, red 5px, yellow 5px, yellow 10px);
```

background: lo standard y lo nuevo

Para establecer el fondo de una etiqueta utilizamos la propiedad **background**. Con ella, podemos establecer tanto un color de fondo como una imagen de fondo a todas las etiquetas, a un elemento identificado con un atributo **id** o **class**.

Por ejemplo, cualquiera de estas reglas harían que el color de fondo fuera blanco:

```
body {background: #FFFFFF;}  
#unelemento {background: #FFFFFF;}  
.unaclase {background: #FFFFFF;}
```

El color, como ya vimos, puede ser expresado en cualquier formato, ya sea una palabra (*white* en el caso de blanco), como hexadecimal (#FFFFFF o #FFF) o como **rgb()**, **rgba()**, **hsl()** o **hsla()**.

Todas estas harían que se mostrase una imagen de fondo donde colocamos la dirección del archivo dentro de **url()**:

```
body {background: url("mi_imagen");}  
#unelemento {background: url("mi_imagen");}  
.unaclase {background: url("mi_imagen");}
```

Por defecto, si usamos **background** con un solo valor, el navegador completará las propiedades restantes ya que esa es una forma de escribir de modo resumido una serie de propiedades que controlan distintos parámetros de manera individual.

Los básicos siguen este orden:

background: color image repeat attachment position;

Por ejemplo:

```
body {background: #FFF url() repeat scroll 0 0;}
```

Cada uno de esos parámetros es una propiedad que puede ser declarada de modo individual:

La propiedad **background-color** define el color de fondo y se debe agregar aún cuando se quiera mostrar una imagen.

Esto es así por dos motivos, el primero es que si la imagen es transparente podemos darle alguna clase de realce colocando un color distinto en el fondo y si no lo es, ese será el color que se mostrará cuando la imagen no se cargue o demore en ser cargada; de ese modo, si hay textos, estos serán legibles.

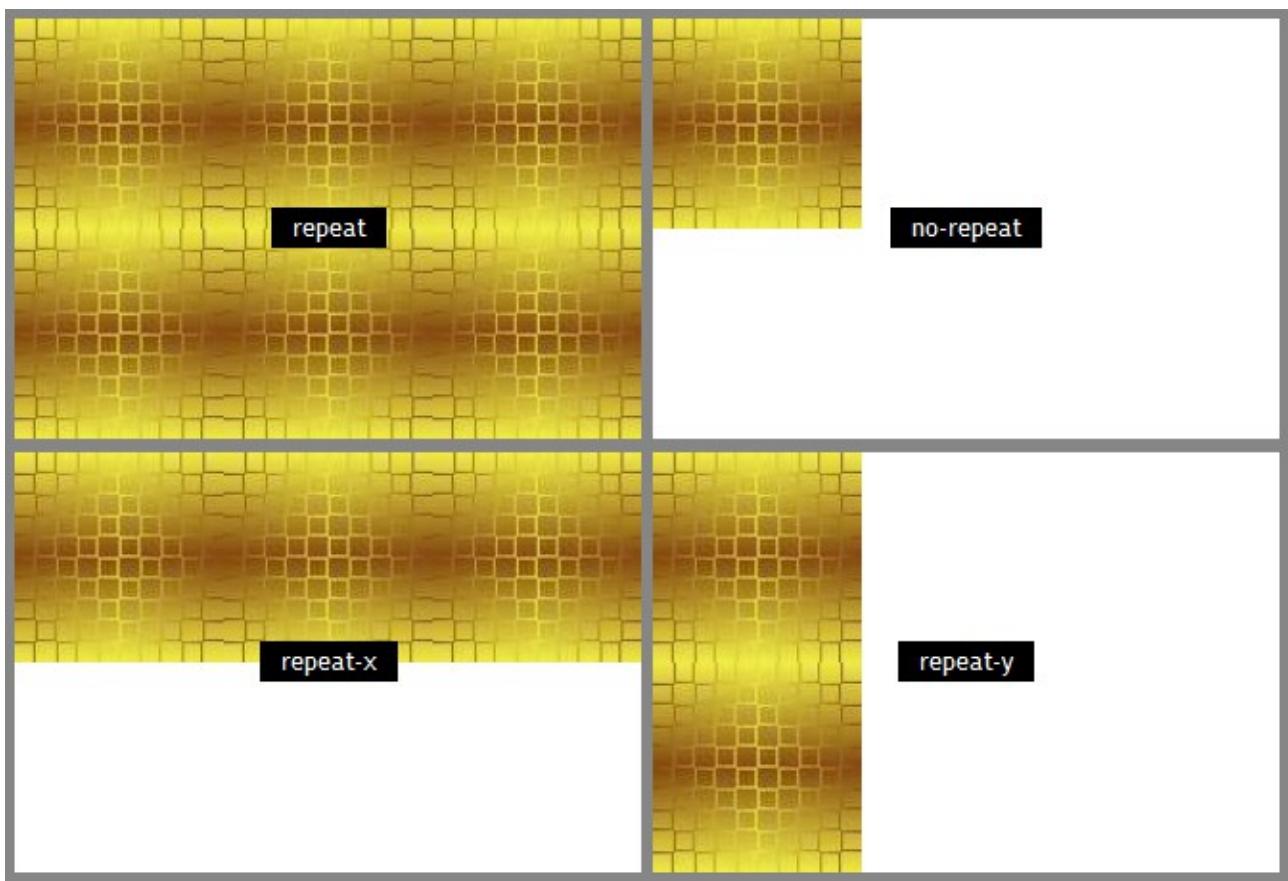
De todos modos, podemos usar la palabra **transparent** para obviarlo.

La propiedad **background-image** indica la dirección URL de la imagen o imágenes a mostrar y las comillas son obligatorias salvo que se usen direcciones absolutas.

Si no queremos usar imágenes podemos indicarlo con `url()` o con la palabra `none`.

La propiedad **background-repeat** establece la forma en que mostrará esa imagen.

Por defecto el valor es `repeat` y se repetirá como un mosaico horizontal y verticalmente hasta ocupar todo el ancho y alto del elemento al que la agregamos; esto es lo que solemos hacer para llenar un determinado espacio con una imagen pequeña. Un valor de `no-repeat` hará lo contrario, la imagen no se repetirá y es lo que usamos para mostrar un logo o cosas similares. Los otros dos valores posibles son `repeat-x` y `repeat-y` que “repiten” la imagen como mosaico pero sólo en una dirección, la primera, horizontalmente y la segunda, verticalmente.



La propiedad **background-attachment** admite dos valores que quizás pueden confundir un poco. El valor `scroll` es el que tiene por defecto y el que usamos habitualmente.

El valor `fixed` hará que la imagen quede fija, que ese fondo, permanezca estático cuando hacemos `scroll` sobre la página y sólo tiene sentido usarlo si el fondo a llenar es más alto que la ventana del navegador, caso contrario, no veremos diferencia alguna.

La propiedad **background-position** esta formada por dos valores que indican en qué posición comenzará a ser mostrada esa imagen de fondo y por defecto es `0 0` o `left top` (la imagen empezará en la parte superior izquierda).

Para indicar esa posición podemos usar palabras porcentajes o pixeles.

La forma más sencilla de asignar una ubicación de fondo es con palabras clave. Las palabras clave son interpretadas como sigue:

top left = left top = 0% 0%
top = top center = center top = 50% 0%
right top = top right = 100% 0%
left = left center = center left = 0% 50%
center = center center = 50% 50%
right = right center = center right = 100% 50%
bottom left = left bottom = 0% 100%
bottom = bottom center = center bottom = 50% 100%
bottom right = right bottom = 100% 100%



Las propiedades básicas para agregar fondos a las etiquetas se han visto ampliadas por la irrupción del CSS3 ya que en los navegadores modernos, hay muchas alternativas extras con las que antes no contábamos ni imaginábamos.

No todas ellas se aplican de modo habitual pero, poco a poco, van extendiéndose y por el momento son tres lo que hace que el esquema básico se amplíe con este orden aunque, estas nuevas propiedades suelen ser definidas de modo individual:

background: color image repeat attachment position / size origin clip;

La propiedad **background-size** permite re-dimensionar las imágenes de fondo, y admite que utilicemos porcentajes, píxeles o palabras especiales

Los valores aceptados son **auto**, **cover** y **contain** que establecen el modo en que la imagen se “expandirá”. Con **cover** ocupará el total, con **contain** lo hará ocupando solo el alto o el ancho con lo que la imagen no se deformará:

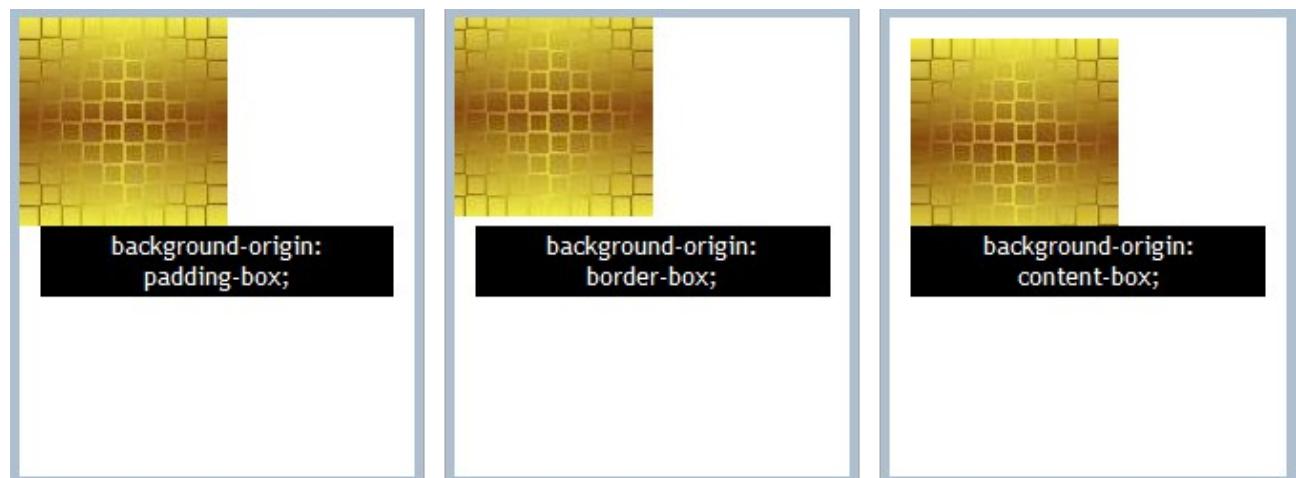


[VER REFERENCIAS \[Los misterios de background-size\]](#)

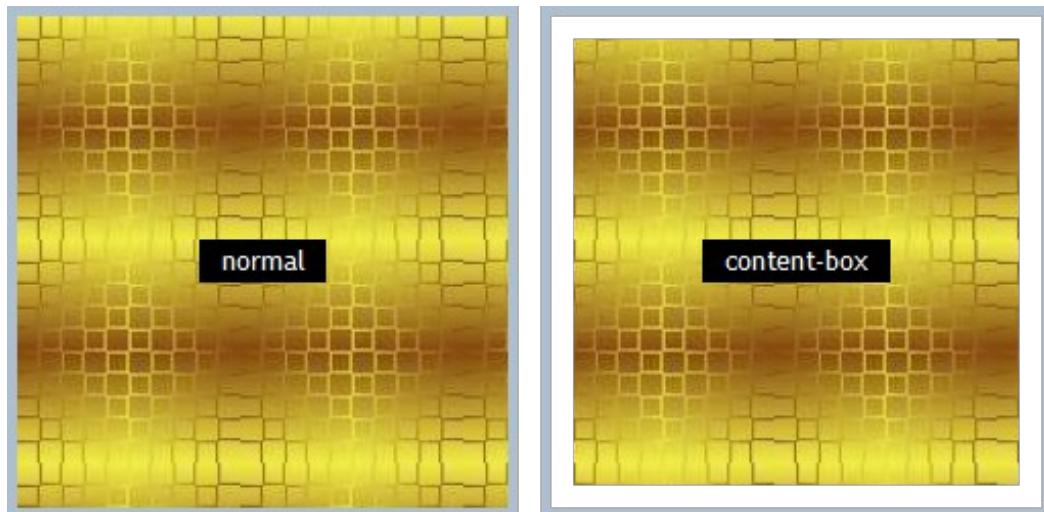
La propiedad **background-origin** define desde dónde comenzará a mostrarse la imagen.

Por defecto, su valor es **padding-box** así que si la etiqueta tienen un *padding* y un borde, la imagen abarcará el *padding* pero no el borde. Usando el valor **border-box**, la imagen abarcará el borde pero no el *padding* (quedará por debajo del borde) y con **content-box** no incluirá a ninguno de los dos:

background-origin: padding-box;
background-origin: border-box;
background-origin: content-box;



La propiedad **background-clip** controla la forma en que se muestran los fondos con relación a sus bordes o *paddings* y puede tener tres valores: **border-box**, **padding-box** o **content-box** que es la que más uso práctico puede tener ya que nos permite establecer *paddings* y no llenarlos con el fondo que es lo que ocurriría normalmente.



VER REFERENCIAS [Los misterios de background-clip]

Tal vez, la característica más interesante de opciones de la propiedad **background** sea la posibilidad de agregar varias imágenes de fondo en un mismo elemento. Para esto no hacen falta propiedades diferentes sino que se usan las misma, separando los valores con comas; por ejemplo:

```
background-color: white;  
background-image: url("imagen_1"), url("imagen_2");  
background-position: 50% 50%, 0% 0%;  
background-repeat: no-repeat, repeat;
```

Cada imagen la controlamos de modo individual y se muestran como capas ordenadas, la primera es la que está más abajo y la última es la que está más arriba por lo tanto debemos tener en cuenta este detalle si es que se superponen.



La máxima cantidad de imágenes a usar no está determinada, en el ejemplo de la izquierda se usan dos y en el de la derecha tres.

[VER REFERENCIAS \[Múltiples fondos con CSS3\]](#)

La propiedad **background-blend-mode** define la forma en que dos imágenes o colores de fondo se “mezclan” cuando están superpuestas.

El valor por defecto es **normal** y las opciones son varias: **color, color-burn, color-dodge, darken, difference, exclusion, hard-light, hue, lighten, luminosity, multiply, overlay, saturation, screen y soft-light**.

Supongamos que tenemos un <div> con dos imágenes de fondo y un texto cualquiera:

```
div {  
    background-image: linear-gradient(to right, yellow, red), url("URL_imagen");  
    background-size: 200px 200px;  
    text-align: center;  
    width: 200px;  
}
```



Si no aplicáramos la propiedad **background-blend-mode** o utilizáramos el valor **normal**, veríamos esto:

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Y esto es lo que pasaría si utilizamos los distintos valores de `background-blend-mode`:

color

`background-color: yellow;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

color-burn

`background-color: yellow;`
`background-blend-mode: color-burn;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

color-dodge

`background-color: yellow;`
`background-blend-mode: color-dodge;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

darker

`background-color: red;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

difference

`background-color: red;`
`background-blend-mode: difference;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

exclusion

`background-color: red;`
`background-blend-mode: exclusion;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

hard-light

`background-color: red;`
`background-blend-mode: hard-light;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

hue

`background-color: red;`
`background-blend-mode: hue;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

lighten

`background-color: red;`
`background-blend-mode: lighten;`
`background-image: url('https://i.imgur.com/3QHgkMn.jpg');`

Placeholder text: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

luminosity

Convoluted text placeholder: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitiation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

multiply

Convoluted text placeholder: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitiation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

overlay

Convoluted text placeholder: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitiation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

saturation

Convoluted text placeholder: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitiation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

screen

Convoluted text placeholder: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitiation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

soft-light

Convoluted text placeholder: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exercitiation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

VER REFERENCIAS [Un poco más allá de las imágenes]

border y outline: los recuadros

Ya vimos que la propiedad **color** permite que establezcamos el color de un elemento (fundamentalmente los textos) y la propiedad **background** permite que definamos el color de fondo.

Ahora le toca el turno a los bordes cuya propiedad genérica es **border** y que es una propiedad muy sencilla aunque posee un sinfín de posibles instrucciones.

Al igual que otras, esta propiedad tiene una definición reducida donde podemos establecer los distintos datos al mismo tiempo:

border: width style color;

Aunque solemos utilizar ese formato, cada una de ellas puede ser definida por separado:

border-width establece el grosor del borde

border-style establece el tipo o estilo

border-color establece el color y por defecto su valor es igual al color del texto

Recordemos que, todo elemento de una página web es un rectángulo y por lo tanto, tiene cuatro bordes así que esta propiedad puede ser aplicada a cada uno de ellos de modo individual.

border-top el borde superior

border-right es el borde derecho

border-bottom es el borde inferior

border-left es el borde izquierdo

Así que para apabullarnos, tenemos las tres primeras combinadas con las cuatro segundas:

border-top-width el grosor del borde superior

border-top-style el tipo del borde superior

border-top-color el color del borde superior

border-right-width el grosor del borde derecho

border-right-style el tipo del borde derecho

border-right-color el color del borde derecho

border-bottom-width el grosor del borde inferior

border-bottom-style el tipo del borde inferior

border-bottom-color el color del borde inferior

border-left-width el grosor del borde izquierdo

border-left-style el tipo del borde izquierdo

border-left-color el color del borde izquierdo

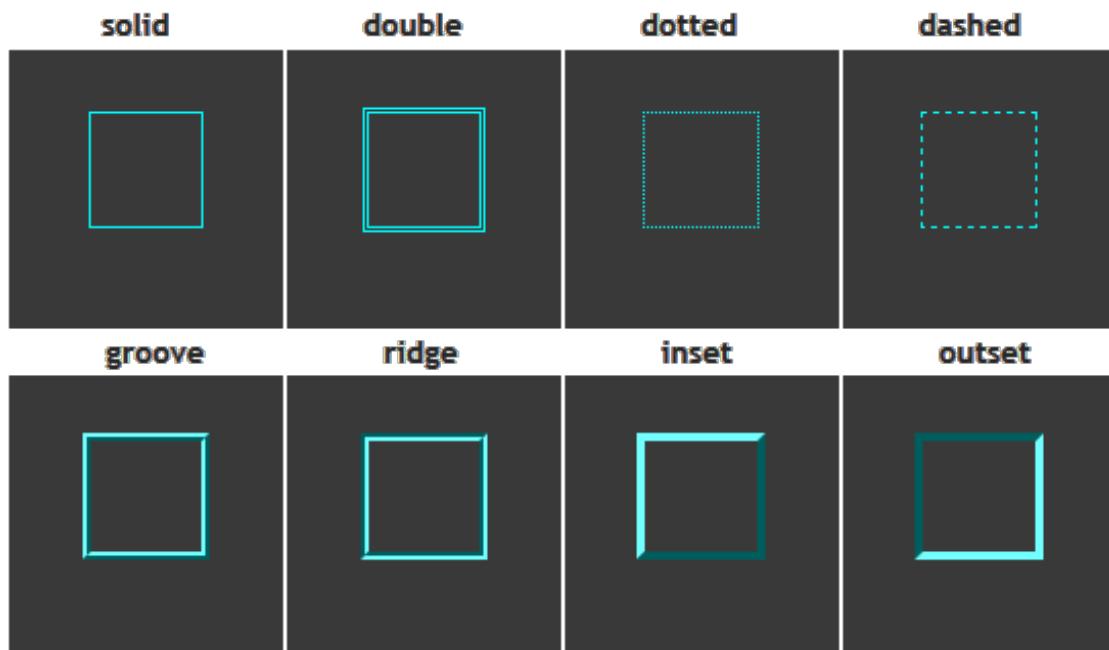
Los valores a utilizar para definir el color de un borde son los mismos que para cualquier otra propiedad.

El valor de **border-width** también es similar a cualquier otra propiedad y podemos elegir la unidad que nos guste aunque la más utilizada es **px** para indicar pixeles pero hay tres palabras alternativas que definen bordes finos, medianos y grueso: **thin**, **medium** y **thick**.

El valor de **border-style** es una palabra que indica el estilo del borde cuyos tipos básicos son **dashed** (guiones), **dotted** (puntos), **solid** (una línea sólida), **double** (dos líneas sólidas); además, los que generan algún efecto que varía con el navegador son **groove**, **inset**, **outset** y **ridge**.

EL valor de **border-style** por defecto de cualquier etiqueta es **none** (que equivale a un grosor de cero) pero, hay casos en que también se utiliza **hidden**.

Hay que considerar que algunos estilos del borde sólo son visibles si el ancho de este es suficiente.



Si utilizamos las propiedades individuales podemos establecer los valores de los cuatro bordes o sólo de alguno de ellos. Estos criterios son válidos para cualquiera de las propiedades de bordes.

Este ejemplo agrega un borde de 5 pixeles a cada uno de los cuatro lados:

border-width: 5px;

Si colocamos cuatro valores, se aplican con un cierto orden: **top**, **right**, **bottom**, **left**:

border-width: 5px 4px 3px 1px;

Si sólo se establece un valor, se aplica a todos los lados. Si se dan dos o tres valores, los valores faltantes se toman del lado opuesto.

Si establecemos tres valores, el orden será el mismo pero el segundo (**right**) se aplicará en ambos bordes verticales (**right** y **left**).

Si sólo definimos dos valores, el primero será el de los bordes horizontales (**top** y **bottom**) y el segundo el de los verticales (**right** y **left**).

Como dijimos, lo mismo ocurrirá con **border-color** y **border-style**, basta recordar que los valores deben estar separados con un carácter espacio.

Es muy importante tener en cuenta que los navegadores agregan los bordes de diferente manera y eso puede causarnos trastornos que cuesta resolver. Algunas de esas diferencias se refieren al aspecto gráfico, pero, lo más significativo se refiere a dónde ubican los bordes.

VER REFERENCIAS [Los bordes ocupan espacio]

El CSS3 nos aporta más alternativas para los bordes. Quizás, la más difundida (porque es la más sencilla) sea **border-radius** que es una propiedad que nos permitirá redondear las esquinas de cualquier elemento.

Tiene las mismas características que la propiedad **border**, puede usarse de manera genérica estableciendo los cuatro bordes al mismo tiempo, establecer las propiedades de cada uno de ellos. La sintaxis es la siguiente:

border-radius: valor;

Donde el valor suele estar expresado en pixeles aunque podría usarse cualquier unidad o porcentajes.

Al igual que con las otras propiedades de bordes, estableciendo un valor definimos los cuatro pero también podemos establecerlos uno por uno:

border-radius: valor valor valor valor;

O utilizar las propiedades específicas:

border-top-left-radius redondeado de la esquina superior izquierda
border-top-right-radius redondeado de la esquina superior derecha
border-bottom-left-radius redondeado de la esquina inferior izquierda
border-bottom-right-radius redondeado de la esquina inferior derecha

Ahora bien, cuando pensamos en esto, nos imaginamos “círculos”, obviamente, la palabra *radius* nos indica eso, el valor que le ponemos es el valor del radio de un círculo; tanto es así que podemos crear un círculo casi perfecto colocando un valor de 50% por ejemplo.

Pero eso no es todo, en realidad, **border-radius** es una propiedad que también permite crear elipses:



Para eso, separamos los valores con una barra inclinada; por ejemplo:

border-radius: 20px / 5px;

En ese caso, el primer valor es el radio horizontal y el segundo, el radio vertical.

Hay una propiedad distinta de **border** que tiene características similares y es la llamada **outline** (contorno).

Seguramente cualquiera que navegue en la web los habrá visto porque se crean automáticamente sobre cualquier etiqueta `<a>` y sobre cualquier elemento que pertenezca a un formulario. En teoría, esto se hace así para “destacarlos” sobre el resto cuando tienen el “foco” (cuando están activos).

Como el contorno está siempre encima, no influye en la posición o tamaño del bloque y es dibujado comenzando junto fuera del límite del borde.

La propiedad **outline** es similar a la propiedad **border** excepto por tres cosas: no ocupa espacio (se dibuja “sobre” el elemento); adopta la forma del elemento y no admite redondear sus bordes.

Al igual que **border**, la propiedad genérica **outline** tiene tres partes:

outline: width style color;

Que pueden usarse por separado: **outline-width** **outline-style** **outline-color**

margin y padding

Hay dos propiedades que establecen la separación de los elementos: **margin** y **padding**.

Si bien parecen lo mismo porque a veces no notamos ninguna diferencia entre usar una u otra, no lo son; **margin** es la separación de un elemento de otro y **padding** es el relleno, la separación entre un elemento y aquel que lo contiene.

Ambas se pueden usar de manera reducida (para definir los cuatro lados):

margin: top right bottom left;
padding: top right bottom left;

O individualmente:

margin-top establece el margen superior

margin-right establece el margen derecho

margin-bottom establece el margen inferior

margin-left establece el margen izquierdo

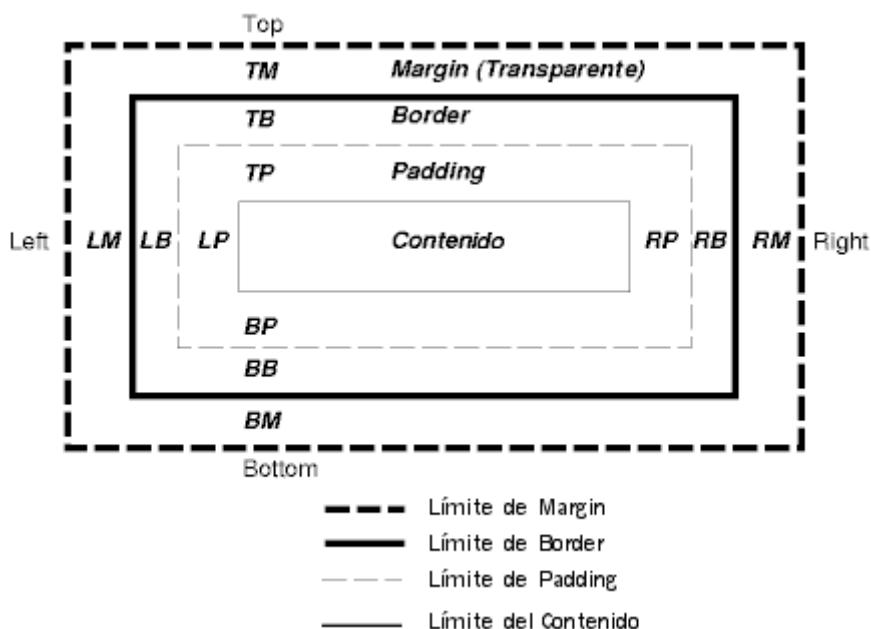
padding-top establece la separación superior

padding-right establece la separación derecha

padding-bottom establece la separación inferior

padding-left establece la separación izquierda

Entonces, **margin** establece la cantidad de espacio entre un elemento y el elemento adyacente y el valor puede ser una longitud (positiva o negativa), un porcentaje o la palabra **auto**. Y **padding** establece la distancia entre los bordes de un elemento y el área que lo contiene y no admite valores negativos.



En ambos casos, el valor por defecto es cero y si no se especifican los cuatro valores, se siguen las mismas reglas que se utilizaban en **border**: si se especifican dos valores, el primero se aplica a **top** y **bottom** y el segundo a **left** y **right**; si se especifican tres valores, el primero se aplica a **top**, el segundo a **left** y **right** y el tercero a **bottom**.

Los valores los sepáramos mediante espacios; por ejemplo:

```
div {margin: 10px 20px;}  
div {padding-left: 5px;}
```

Como se ve, las reglas de sintaxis son muy simples pero, en la práctica, estas dos propiedades pueden ser confusas.

El primer dato a tener en cuenta es que en los elementos de tipo **inline** (por ejemplo ****), los márgenes y *paddings* superior e inferior no tienen efectos prácticos, sólo son visibles en los elementos de tipo bloque (por ejemplo **<p>**).

El segundo dato a recordar es que el **padding** o relleno, es la cantidad de espacio a insertar entre el contenedor y su margen o, si hay un borde (propiedad **border**) la distancia entre el objeto y el borde.

Veamos. Cada elemento de una página web es un rectángulo que tiene un contenido (textos, imágenes, etc.) y una serie de áreas opcionales a su alrededor (**border**, **margin** y **padding**) por lo tanto, el tamaño final depende de todas y cada una de esas propiedades.

En términos generales, el ancho es la suma de los márgenes, los bordes y los rellenos izquierdos y derechos más el ancho del contenido y la altura es la suma de los márgenes, los bordes y los rellenos superiores e inferiores más la altura del contenido.

Veamos un ejemplo; supongamos que tenemos dos etiquetas **<div>**, una donde establecemos un valor para **margin** y otra donde establecemos un valor para **padding**:

```
<div style="margin:20px; padding:0px;">un texto</div>  
<div style="margin:0; padding:20px;">un texto</div>
```

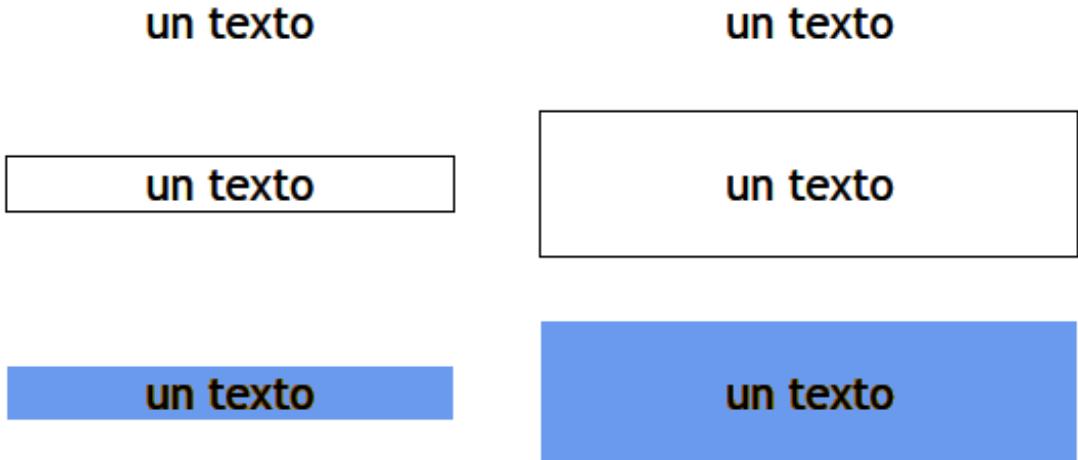
Si vemos eso en el navegador, no notaremos diferencia alguna. Ambas parecen iguales.

Entonces, aparentemente, usar **margin** y usar **padding** es lo mismo pero NO ES LO MISMO y es fácil de ver apenas le agregamos un borde:

```
<div style="border:1px solid #000; margin: 20px; padding: 0px;">un texto</div>  
<div style="border:1px solid #000; margin: 0px; padding: 20px;">un texto</div>
```

O un color de fondo:

```
<div style="background-color: #6495ED; margin: 20px; padding: 0px;">un texto</div>  
<div style="background-color: #6495ED; margin: 0px; padding: 20px;">un texto</div>
```



Como cada etiqueta es un rectángulo, esos rectángulos se pueden separar unos de otros utilizando la propiedad **margin**. El **padding** no separa un rectángulo del otro sino que separa el contenido de ese rectángulo de su borde (sea este visible o no). En resumen:

margin es externo; **padding** es interno
margin no cambia el tamaño del elemento, **padding** si

No hay una regla exclusiva para todos los casos posibles pero, en general, si un elemento no tiene bordes ni fondo, basta definir una cualquiera de esas propiedades y no es necesario agregar ambas. Esto, es algo bastante común y confunde mucho ya que se generan espacios vacíos que se van sumando. Lo mejor es eliminar cuanto **margin** y **padding** exista y luego, agregarlos uno por uno, tratando de controlar esas separaciones y de ese modo, evitar que eso que creemos que tiene cierto tamaño, se vea cortado, apretujado o demasiado separado sin que uno sepa bien por qué ya que, incluso, una etiqueta vacía (sin contenido) puede generar espacios, separaciones que no sabemos de dónde salen.

¿Hay una forma sencilla de “ver” el tamaño real de algo? Si no queremos agregarle un color de fondo se puede utilizar la propiedad **outline** que agregará un borde que no ocupará espacio y por lo tanto, el elemento tendrá el tamaño real y podremos ver el lugar que realmente ocupa dentro de la página. La regla de oro es esta:

ancho = border-left + padding-left + width + padding-right + border-right
alto = border-top + padding-top + height + padding-bottom + border-bottom

Pero el oro es uno de esos materiales impredecibles y esa regla es relativa ya que hay otra propiedad que la modifica de manera sustancial y es una regla que funciona en cualquier navegador moderno.

Sabemos que las propiedades CSS tienen siempre un valor por defecto así que si no las colocamos, los navegadores usan esas definiciones. La gran mayoría de ellas, sobre todo las más sofisticadas, no influyen en nada, son siempre valores nulos o no nos damos cuenta de su existencia. Este es el caso de la propiedad **box-sizing**.

box-sizing: valor;

El valor por defecto es una palabra: **content-box** que indica que el ancho y alto de algo no incluye bordes, *paddings* ni márgenes, sólo su contenido. Pero admite otro valor: **border-box** que establece que el ancho y alto incluyen al borde y *paddings*.

VER REFERENCIAS [margin versus padding]

Un detalle importante cuando se usan márgenes es que muchas veces, estos se solapan creando cierta confusión a la hora de ver el resultado.

Esto es muy común cuando dos elementos adyacentes (uno debajo del otro) tienen márgenes superiores e inferiores; el resultado final no es la suma del margen inferior del elemento superior y el margen superior del elemento inferior sino sólo uno de ellos, el que tenga el mayor valor:

```
<p style="margin: 20px">texto superior</p>
<p style="margin: 10px">texto inferior</p>
```

Esto nunca se aplica a elementos flotantes o que tienen posiciones absolutas.

Alineación

La alineación de las cosas es algo que siempre suele provocar confusiones, sobre todo cuando recién se comienza a meter mano en una página web. Si queremos que algo se vea a la derecha, a la izquierda o en el centro, parecería que no siempre debemos usar las misma propiedades, a veces usamos unas a veces otras y a veces no podemos.

En realidad, no es algo tan complejo si se entiende que una cosa es el contenedor y otra cosa es el contenido. En una etiqueta sencilla como esta:

```
<p>el texto</p>
```

La etiqueta en si misma es el contenedor y el texto es el contenido; podemos alinear ese texto utilizando **text-align** pero, la etiqueta **<p>** en si misma, no se “mueve”, ocupa todo el ancho de la ventana aunque el texto sea corto.

De todas formas, la forma más común de alinear el contenido es utilizando **text-align** aunque su nombre puede llevar a confusión ya que no es algo que sólo se aplica a los textos sino a cualquier otro contenido como una imagen:

```
<div style="text-align: center;">
    
</div>
```

La propiedad **text-align** admite ciertas palabras como valores:

center centra el contenido

left alinea el contenido a la izquierda

right alinea el contenido a la derecha

justify alinea el contenido de modo justificado completando las líneas con espacios

Podría decirse que el problema más común es centrar algo y para eso debemos tener claro un concepto: no todos los elementos (etiquetas) de una página web pueden alinearse; sólo aquellos llamados “bloques”.

¿Qué pasaría si en el ejemplo anterior colocáramos la propiedad en la etiqueta ****? Nada, no produciría ningún efecto porque la etiqueta **** es de tipo **inline**; lo mismo ocurriría si usáramos **<a>** o ****.

Algo más: ¿qué pasaría si en el ejemplo anterior el **<div>** estuviera dimensionado de tal forma que fuera más angosto que la página? Probemos colocándole un color de fondo para “verlo”:

```
<div style="background-color: yellow; text-align: center; width: 300px;">
    
</div>
```

La imagen estaría centrada pero el `<div>` no ¿Cómo lo centramos? Para eso, usamos la propiedad **margin** a la que le ponemos un valor especial: **auto** pero, siempre y cuando esté definida también la propiedad **width** ya que si no lo está, el ancho por defecto es todo el ancho disponible.

Ahora sí, podemos centrar el bloque y el contenido:

```
<div style="margin: 0 auto; text-align: center; width: 300px;">  
      
</div>
```

Alinear algo verticalmente no es tan sencillo como parecería aunque existe la propiedad **vertical-align** que determina la forma en que alineará un elemento *inline* con respecto a su contenedor:

vertical-align: valor;

Donde valor puede ser una longitud, un porcentaje o una palabra clave:

baseline alinea la base del elemento con la del contenedor y es el valor por defecto
middle centra el elemento verticalmente

top alinea la parte superior del elemento con la parte superior del contenedor

bottom alinea la parte inferior del elemento con la parte inferior del contenedor

text-top alinea la parte superior con la parte superior de la fuente del contenedor

text-bottom alinea la parte inferior con la parte inferior de la fuente del contenedor

sub alinea la base del elemento con la línea *subscript* del contenedor

super alinea la línea base del elemento con la línea *superscript* del contenedor

A diferencia de la alineación horizontal, la propiedad se coloca en el contenido:

```
img {vertical-align: bottom;}
```

VER REFERENCIAS [Alinear verticalmente]

Dimensiones y visualización

Empecemos por el principio ¿Qué es un `<div>`? Pués, nada más simple; las etiquetas `<div>` son rectángulos que, por defecto, no tienen ninguna propiedad; es lo que suele llamarse una etiqueta contenedora es decir, un lugar donde podemos agregar cosas. En realidad, cuando digo que no tiene propiedades, me refiero a propiedades extras ya que estas etiquetas tienen algunas características especiales, se “separan” de lo que está arriba y de lo que está debajo de ellas (por eso las llamamos etiquetas de bloque) y tienen un ancho que es igual al ancho total y una altura que depende de su contenido.

Este dato es importante ¿qué es el ancho total? El ancho básico es el de la pantalla del dispositivo, el `<body>` de nuestro sitio pero, como podemos poner una etiqueta dentro de otra, el ancho de cada rectángulo depende del ancho del rectángulo que lo contiene.

Si escribimos esto:

```
<div>
  <div>
    <div>
      <div>
        <div>acá va el contenido</div>
      </div>
    </div>
  </div>
</div>
```

No veremos nada distinto a si escribimos esto:

```
<div>acá va el contenido</div>
```

Ya que esos rectángulos no tienen propiedades y los tamaños no están definidos así que podemos tener cientos y cientos entrelazados sin que eso cambie absolutamente nada.

Sin embargo, hay propiedades que nos permiten definir el tamaño de esos elementos de bloque. Para eso tenemos a `width` y `height` que establecen su ancho y su alto.

```
div {height: valor; width: valor;}
```

Por defecto, el valor de ambos es auto pero podemos indicar cualquier otro, expresándolo en cm, mm, in, pt, pc, px, em, ex o como un porcentaje.

El valor nunca puede ser negativo y en muchos casos (por ejemplo, las imágenes), indicando uno de ellos, el otro adopta el valor proporcional correspondiente y no es necesario expresarlo.

Asociadas con estas, hay otras propiedades que nos permiten establecer los anchos y altos máximos y mínimos de los elementos:

max-width: valor;

min-width: valor;

max-height: valor;

min-height: valor;

Otras etiquetas como ``, `<a>`, `` o `` no son bloques, sólo ocupan el ancho de su contenido y son las llamadas etiquetas *inline* y en ese caso, esas propiedades no tienen efecto.

Sin embargo, cualquier etiqueta *inline* puede ser convertida en una etiqueta de bloque y viceversa. La propiedad **display** establece cómo será mostrado un elemento.

display: valor;

Donde los valores son palabras predefinidas:

block el elemento se muestra como un bloque

inline el elemento tiene las dimensiones de su contenido

inline-block similar al anterior pero el contenido se muestra como bloque y los elementos adyacentes se muestran en la misma línea

none el elemento permanece oculto

list-item el elemento muestra su contenido separado como las etiquetas de listas

inline-table el elemento se muestra como una tabla

Otra serie de posibles valores asemejan los elementos a las distintas etiquetas de las tablas:

table hace que el elemento se comporte como `<table>`

table-cell hace que el elemento se comporte como `<td>`

table-row y **table-row-group** hacen que el elemento se comporte como `<tr>`

table-column hace que el elemento se comporte como `<col>`

table-column-group hace que el elemento se comporte como `<colgroup>`

table-header-group hace que el elemento se comporte como `<thead>`

table-footer-group hace que el elemento se comporte como `<tfoot>`

table-caption hace que el elemento se comporte como `<caption>`

Hay otros valores posibles: **marker**, **compact** y **run-in** que difícilmente tienen uso práctico,

La propiedad **visibility** nos permite mostrar u ocultar un elemento y parece ser lo mismo que **display=none** pero es fundamentalmente distinta.

Cuando usamos **display=none** el elemento y su contenido permanecen ocultos por completo pero con **visibility** el elemento no “desaparece” de la página, el contenido no se ve pero sigue ocupando un espacio en la página.

Tiene sólo dos valores posibles: **visible** (que es el estado normal de cualquier etiqueta) y **hidden**.

Ya se ha dicho muchas veces pero es un concepto sobre el que hay que insistir. Todos los elementos de una página web son rectángulos. Cuando agregamos alguno, lo que hacemos es decirle al navegador que cree un área rectangular con un cierto contenido. Algunos de esos elementos son bloques porque se separan del elemento inferior con un salto de línea (se apilan) y otros no lo son (*inline*), no se separan, se colocan uno al lado del otro.

Por ejemplo, si agregamos dos imágenes y escribimos:

```

```

Se mostrarán una al lado de la otra porque `` es un elemento *inline*.



Sin embargo, si colocamos cada una dentro de una etiqueta `<div>` o `<p>`, se mostrarán una debajo de la otra porque son elementos tipo bloque (*block*).

```
<div></div>
<div></div>
```



Entonces, usando la propiedad `display`, podemos modificar el comportamiento de un elemento y hacer que una etiqueta que normalmente es *inline* se comporte como bloque:

```

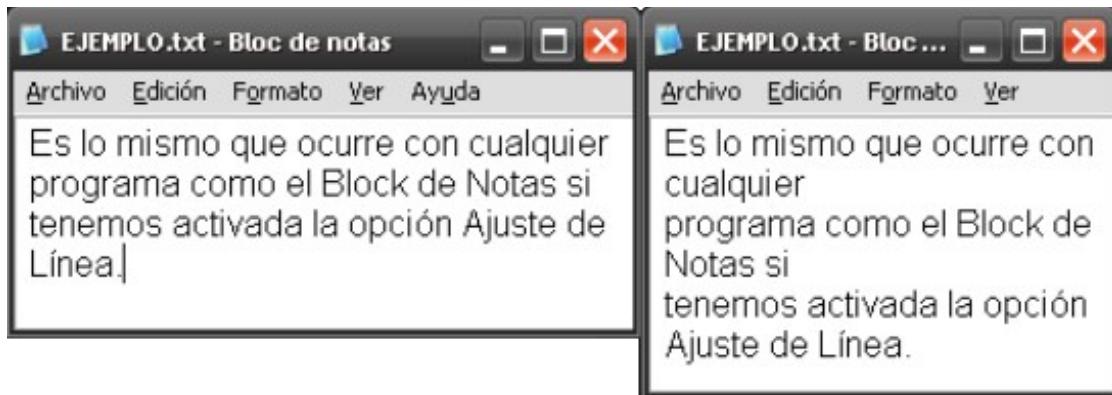

```

Y viceversa:

```
<div style="display:inline;"></div>
<div style="display:inline;"></div>
```

Esto no es cierto, puede decir alguien que usó dos imágenes y, sin agregar propiedades, aparecieron una debajo de otra ¿hay algún un error?

No, no es un error, lo que puede ocurrir es que, si el espacio necesario para contener ambas imágenes es menor que la suma del ancho de las mismas, el navegador empuja la segunda hacia abajo.



Pero ¿y si quiero que se superpongan? ¿acaso en una página web dos elementos pueden ocupar el mismo espacio contradiciendo las leyes de la física?

La respuesta es sí.

[VER REFERENCIAS \[Sobre IDs y bloques ocultos\]](#)

Posición

En CSS hay propiedades que controlan la posición de los elementos y con ellas podemos modificar los valores por defecto:

La propiedad **top** establece la posición superior.

La propiedad **right** establece la posición derecha.

La propiedad **bottom** establece la posición inferior.

La propiedad **left** establece la posición izquierda.

Pero, para que esto funcione, debemos incluir una propiedad extra llamada **position** que puede tener distintos valores:

static es el valor por defecto de toda etiqueta

absolute indica que la posición es absoluta respecto al contenedor

relative indica que la posición es relativa a su posición normal

fixed indica que la posición es relativa a la ventana y no se mueve al hacer *scroll*

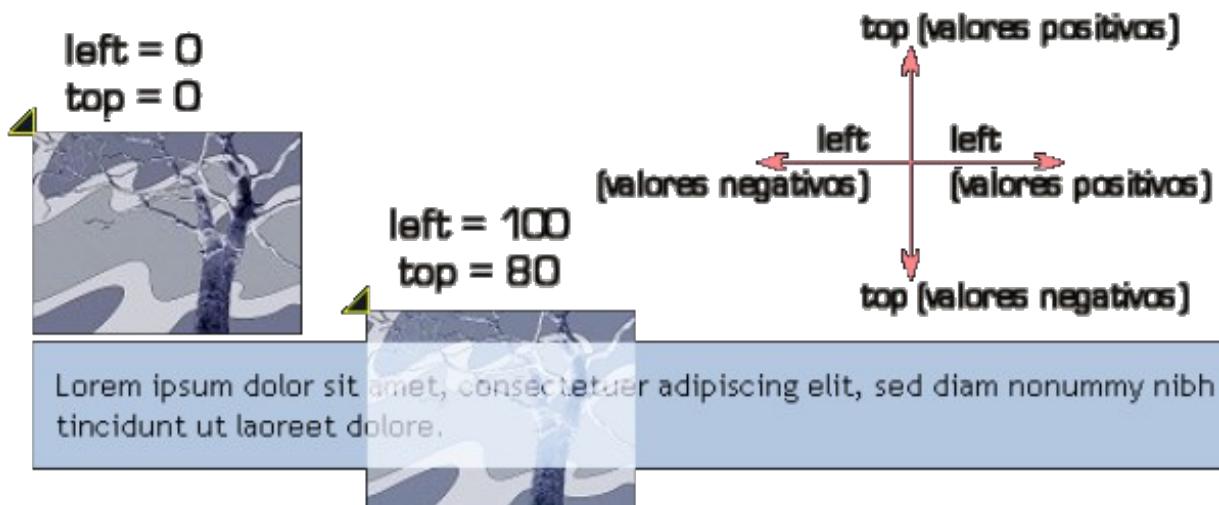
Parece confuso pero es más simple de lo que parece. Veamos este código:

```

<div style="background-color:LightSteelBlue; color:black;">contenido</div>
```

Mostrará la imagen arriba y un recuadro con texto abajo de ella.

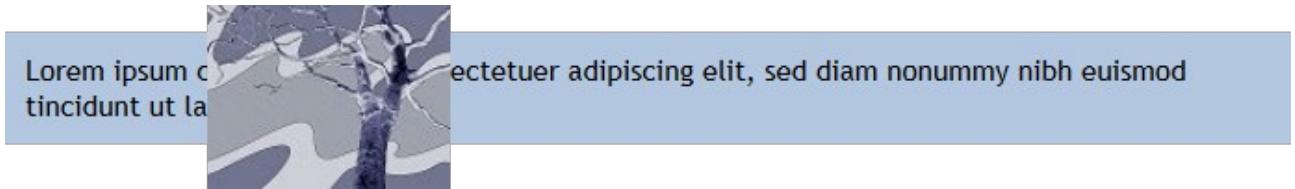
La posición se dice relativa al lugar en donde el elemento se ubica por defecto y es su coordenada superior izquierda. Quiere decir que si queremos que, en lugar de aparecer donde se muestra normalmente, la queremos poner más abajo y a la izquierda, bastaría establecer esas propiedades con el valor respectivo:



El código sería este:

```

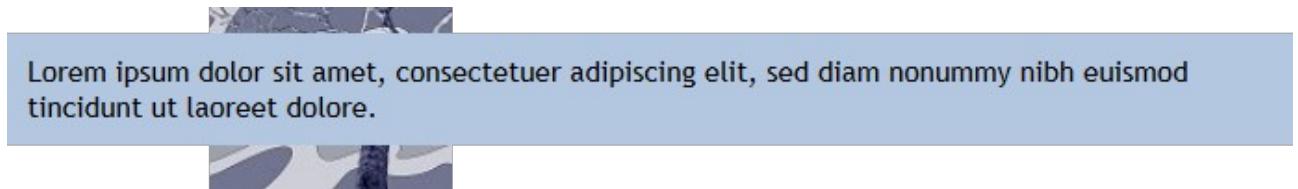
<div style="background-color: LightSteelBlue; color: black;">contenido</div>
```



Establecemos la posición de `` como relativa para poder “moverla” pero, ¿qué pasa si el `<div>` también tiene la propiedad **position**?:

```

<div style="position: relative; background-color: LightSteelBlue; color: black;">contenido</div>
```



Si ambos elementos tienen el valor **relative**, se muestran como lo harían normalmente, siguiendo el orden en que fueron creado: se pone la imagen, se mueve, se pone el bloque de texto, como la imagen está abajo, queda parcialmente tapada. Si sólo el elemento que vamos a re-posicionar posee esa propiedad **relative**, tiene “prioridad” sobre los otros y, por lo tanto, aunque se haya dibujado primero, aparece en primer plano, por delante del texto.

En resumen, la clave es definir un contenedor externo con **position:relative** y de ahí en más, cualquiera sea su contenido, basta definirlo con **position:absolute** y establecer las propiedades **top**, **right**, **bottom** y/o **left** para “re-ubicarlo” con valores que pueden ser expresados en cualquier unidad o porcentaje, tanto positivos como negativos.



Por defecto, todo elemento se encuentra en **top:0px** y **left:0px**, es decir, se ubica en la parte superior izquierda del contenedor.

Avancemos un poco más.

Creemos un código con cuatro imágenes posicionadas con diferentes valores para **left** y **top**:

```




```



Las coordenadas **left** y **top** mueven la imagen pero la primera de ellas quedará debajo y la cuarta arriba, es decir, cada una se superpondrá con la otra porque ese es el orden que le damos en el código.

El navegador lee la primera, la muestra, lee la segunda, la coloca encima, y así sucesivamente. En términos simples, las apila; en términos sofisticados, les da un número de orden en el *stack*; un número menor implica una posición inferior (está abajo), un número mayor implica una posición superior (está arriba).

un valor menor implica que el elemento
está más cerca del fondo



un valor mayor implica que el elemento
está más lejos del fondo

Esta posición puede ser modificada utilizando CSS, para eso, existe una propiedad específica llamada **z-index** y que, al igual que las otras, sólo funciona cuando se establece la propiedad **position** como **absolute** o **relative**.

Con esta propiedad, podemos determinar el orden en que se apilan los elementos y acepta valores tanto positivos como negativos:

Usemos el mismo código pero agreguemos valores la propiedad **z-index** a la imagen 2 para que aparezca por encima de las otras:

```




```



¿Por qué un valor de 1? Simplemente porque si la propiedad no está establecida (si no la ponemos) no se tiene en cuenta, es como fuera cero.

Sólo debe recordarse que el valor de **z-index** es un numero de orden y, dicho muy livianamente, indica cuál está encima de cuál.

VER REFERENCIAS [Las posiciones absolutas son relativas a “algo”]

Cuando las cosas se desbordan

Como toda etiqueta es un rectángulo, tiene un ancho y un alto. Cuando colocamos algo adentro (sea un texto, una imagen, un vídeo, etc), ese “contenedor” se ajusta automáticamente; si el contenido es “más grande” que el contenedor, este último se amplia. Esa es una de las características más interesantes del diseño web, lo que lo hace flexible y hasta sencillo ya que uno no debe preocuparse por detalles tales como calcular tamaños.

Esto es así porque por defecto hay cuatro propiedades CSS que lo indican:

```
height: auto;  
overflow: visible;  
width: auto;  
word-wrap: normal;
```

Por lo tanto, a menos que nosotros las cambiemos, esas serán las propiedades que tendrá cualquier etiqueta aunque es común que modifiquemos alguna, por ejemplo definiendo un ancho.

Pero ¿qué ocurriría si el tamaño del contenedor es menor que el contenido?

Como el contenido “no entra”, el bloque se re-dimensiona, ignorando el tamaño que establecimos aunque le hayamos “ordenado” que respete nuestras medidas; para colmo, es posible que se agreguen unas barras de desplazamiento en uno o ambos sentidos.

La propiedad **overflow** permite controlar qué se va a hacer cuando un elemento sobrepasa el área de su contenedor:

```
overflow: valor;
```

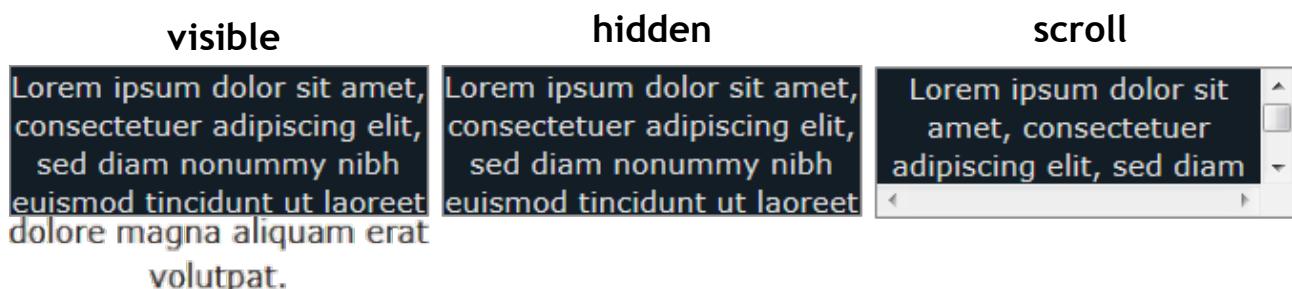
Y tiene cuatro valores posibles

auto se agregará una barra de desplazamiento donde sea necesario

hidden el contenido es recortado

visible el contenido no es recortado (es el valor por defecto)

scroll el contenido es recortado y el navegador mostrará barras de desplazamiento



Además, la propiedad `overflow` tiene dos variantes con las cuales es posible definir los desbordes en cada una de las direcciones:

`overflow-x` establece la forma en que se manejará si se excede el ancho
`overflow-y` establece la forma en que se manejará si se excede el alto

Siempre es bueno tener claro este problema en todos los contenedores cuyo ancho deseamos controlar pero responder cómo manejar esos posibles “desbordes” depende de cada caso en particular.

Podría decirse que hay dos situaciones diferentes, una afecta a los textos y otra afecta al resto del contenido (imágenes, vídeos, etc). Para estos últimos, lo razonable es establecer en ellos la propiedad `overflow:hidden`; ya que de ese modo impediremos que ciertas cosas se solapen o aparezcan ocupando lugares indebidos.

En el caso de los textos eso no es del todo bueno ya que si usamos esa propiedad, se cortarán y no podrán leerse.

Para evitar esto, existe otra propiedad que deberíamos utilizar y que le indica al navegador que, cuando hay un texto demasiado largo, simplemente, lo “corte” en líneas.

Esa propiedad es `word-wrap` y el valor a usar en este caso es `break-word` ya que el otro valor posible es `normal` que es el valor por defecto.

`word-wrap` es una propiedad que en realidad, fue inventada por Microsoft para sus navegadores pero que rápidamente fue adoptada por todos los demás. Aún discuten si cambiarle el nombre a `overflow-wrap` pero eso ya es pura semántica.

Asociado con los desbordes, seguramente habrán notado que en algunas etiquetas como `<textarea>` se ve en la parte inferior derecha un símbolo que nos permite cambiar su tamaño, arrastrando y soltando el puntero del ratón del mismo modo que usualmente re-dimensionamos cualquier otra cosa.

Esto, que es tan útil, no es otra cosa que la incorporación a los estilos por defecto que tienen ciertas etiquetas en estos navegadores, de una de las nuevas propiedades del CSS3 llamada `resize` porque no está limitada a los formularios sino que se la puede utilizar con otro tipo de etiqueta de bloque como un `<div>` siempre que esa etiqueta, además, posea una propiedad `overflow`, distinta de `visible`.

La propiedad `resize` puede tener uno de cuatro valores: `none` es el valor por defecto, `both` permite re-dimensionar en ambos sentidos y `horizontal` o `vertical`, sólo en uno de ellos.

Si no quisiéramos permitir ese re-dimensionamiento, deberíamos indicarlo en una regla de estilo:

`textarea {resize: none;}`

La re-dimensión sólo tiene como límite los bordes del contenedor, en este caso, la re-dimensión horizontal está contenida por los bordes de la entrada en si misma pero, la vertical no tiene límites. Si quisieramos dárselos, es decir, permitir que se re-dimensione sólo hasta cierto tamaño y no más, podríamos usar las propiedades **max-width** y **max-height**, indicando en ellas, los valores correspondientes, lo mismo que los mínimos con las propiedades **min-width** y **min-height**.

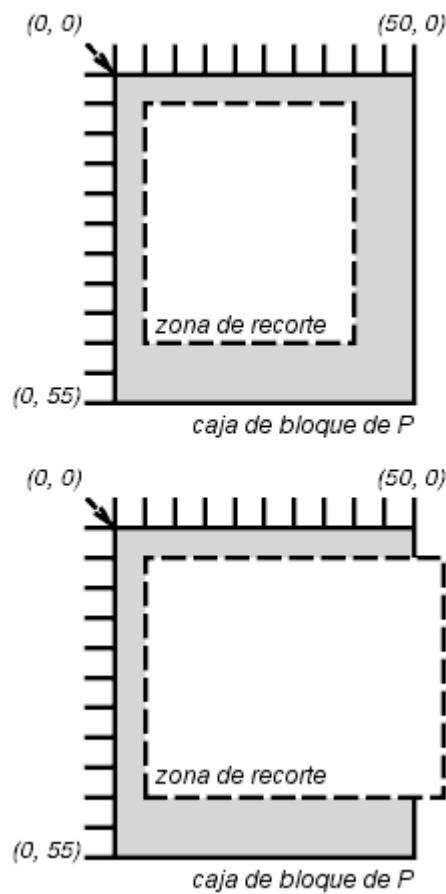
En CSS también hay una propiedad que permite recortar una parte de un elemento cuando se produce un desbordamiento y con la propiedad **overflow** se especificó algún tipo de recorte.

La propiedad **clip** nos permite establecer el tamaño y la forma de ese recorte y tiene dos valores: **auto** que es el valor por defecto y la función **rect()** que es donde debemos establecer las coordenadas para el recorte:

clip: rect(top right bottom left);

Para establecer los valores **top**, **right**, **down**, **left** se debe considerar 0,0 al ángulo superior izquierdo del elemento, de tal manera, un valor en **top** “cortará” todo lo que esté por encima, uno en **bottom**, todo lo que esté por debajo, un valor en **left** todo lo que esté a la izquierda y un valor en **right** todo lo que esté a la derecha.

Actualmente, en CSS, las zonas recortadas siempre son rectangulares pero se estima que en el futuro se admitirá otro tipo de “recortes”.



Las fuentes de los textos

Las propiedades que nos permiten personalizar los textos de una página web son muchas, algunas las usamos habitualmente pero hay otras que son casi desconocidas.

Para empezar, hay una resumida que ayuda a definir varias propiedades de manera simultánea: la propiedad **font** donde cada valor se separa del otro mediante un espacio:

font: style variant weight size height family

Por ejemplo:

font: italic small-caps bold 12pt serif;

La propiedad **font-family** define el tipo de fuente, es decir el tipo de letra a usar; por ejemplo:

font-family: arial, sans-serif;

font-family:Broadway,Arial,Helvetica,sans-serif;

Sin embargo, hay que recordar que si bien podemos indicar cualquiera, lo que verán los visitantes dependerá de sus sistemas operativos, del navegador que utilicen y de las fuentes que tengan instaladas. Es común ver que se definen fuentes exóticas porque, claro, quien diseña la página las ve pero eso no significa que lo hará el resto de los usuarios.

Esto, no es un problema si es que somos conscientes de ello. Una forma de minimizar los riesgos es utilizar fuentes alternativas que se agregan, separándolas con comas y siempre poner una de las llamadas familias genéricas (*serif*, *sans-serif*, *cursive*, *fantasy*) como la última.

Una “familia” es un grupo de fuentes de características similares. Cada “miembro” de la familia puede ser cursiva, negrita, condensada, etc. Las familias de fuentes pueden ser agrupadas en categorías: con o sin *serif*, con o sin caracteres espaciados de modo proporcional, etc.

Los nombres compuestos (que tienen varias palabras) se deben escribir entre comillas pero cualquier nombre de una fuente válida es aceptado; por ejemplo: *Times*, *Helvetica*, *Zapf-Chancery*, *Western*, *Courier*.

También pueden usarse nombres genéricos: *serif*, *sans-serif*, *cursive*, *fantasy* o *monospace*.

La propiedad **font-size** define el tamaño de la fuente de texto.

font-size: valor;

Los valores que podemos utilizar son muy variados; pueden ser unidades fijas o proporcionales como cm, mm, in, pt, pc, px, em o ex, un porcentaje o palabras claves tales como **xx-small**, **x-small**, **small**, **medium** (este es el valor por defecto), **large**, **x-large**, **xx-large**. También se puede utilizar **smaller** o **larger** para indicar una fuente de un tamaño inmediatamente menor o inmediatamente mayor.

Aunque podemos elegir cualquier unidad, las absolutas como puntos, centímetros o pulgadas deberían usarse razonablemente debido a su poca capacidad de adaptarse a diferentes ambientes de navegación (pueden ser muy pequeñas o muy grandes para un usuario) dependiendo del dispositivo.

La propiedad **font-style** establece el estilo de la fuente y tiene tres valores posibles: **normal**, **italic** y **oblique**.

La **italica** es la versión cursiva del tipo normal. La **oblicua** es una versión inclinada del tipo normal y comúnmente se usa con las fuentes de tipo *sans-serif*.

La propiedad **font-variant** establece si la fuente se muestra en modo mayúsculas pequeñas. Es decir, todos los caracteres se convierten en mayúsculas pero el tamaño es menor que el tamaño de la fuente normal. Sólo tiene dos valores: **normal** o **small-caps**.

La propiedad **font-weight** establece el grosor de las fuentes (en negrita o **bold**). Los valores posibles son: **normal** , **lighter**, **bold** y **bolder**. También pueden usarse algunos valores numéricos: 100, 200, 300, 400, 500, 600, 700, 800 y 900. Los valores **bolder** y **lighter** son relativos a la fuente original, mientras que los otros valores son pesos de fuente absolutos.

Ya que no todas las fuentes tienen nueve posibles pesos de visualización, algunos de los pesos pueden agruparse en la asignación. Los valores 100 a 900 indican un ancho que es cada vez más oscuro. Un tipo **normal** se corresponde con el valor 400, **bold** con el valor 700.

Las variaciones dependen del tipo de fuente y muchas veces es difícil notar las diferencias.

La propiedad **font-size-adjust** también establece el tamaño de la fuente; en realidad, el tamaño y la legibilidad de una fuente depende menos del valor dado a **font-size** que de la proporción entre su ancho y su alto. Esa relación (tamaño de la fuente dividido por la altura de la letra x), es llamada *x-height*. Cuanto más grande sea ese valor menos legible será la fuente si se trata de fuentes pequeñas. Los valores posibles son: **none** o un número que especifica esa relación. Por ejemplo

font-size-adjust: 0.8;

La propiedad **font-stretch** permite condensar o estrechar una fuente seleccionando un tipo normal, condensado o expandido de una familia de fuentes.

Los valores posibles son: **ultra-condensed**, **extra-condensed**, **condensed**, **semi-condensed**, **normal**, **semi-expanded**, **expanded**, **extra-expanded** o **ultra-expanded**.

También pueden usarse las palabras clave **wider** para modificar el valor al siguiente más expandido y o **narrower** para seleccionar al siguiente más condensado.

Las fuentes de textos que podemos utilizar en una página web son limitadas porque lo que se verá no es necesariamente lo que nosotros establecemos sino aquello que permite el navegador de los visitantes. Podemos decir que un párrafo tendrá la fuente más exótica que se nos ocurra pero, si quien mira la página no tiene esa fuente en su dispositivo, no la verá.

Como vimos antes, es por eso que solemos definirlas con varios tipos o familias, para que, si no hay una, se cargue otra pero, a menos que usemos las fuentes “seguras”, el resultado es imprevisible.

Sin embargo, hay un método para incrustar fuentes externas utilizando CSS. De algún modo, es similar a insertar *scripts*, se requiere uno (o varios) archivos externos que se cargarán junto con el resto de la página lo que hará que el tiempo de carga se incremente, a veces, de modo sustantivo. A esto, se le suma una limitación más. Muchas de las fuentes están protegidas por derechos de *copyright* y están inhabilitadas para ser insertadas en la web. De todos modos, la sintaxis es esta:

```
@font-face {  
    font-family: "nombre_fuente";  
    src: url(archivo);  
}
```

Un detalle importante es que para que esto funcione correctamente, debemos tener distintos archivos porque los distintos navegadores sólo aceptan determinados formatos que, básicamente son: EOT, SVG, TTF/OTF (TrueType y OpenType), WOFF/WOFF2 (Web Open Font Format)

Las fuentes de tipo TTF/OTF son aceptadas por todos; las fuentes WOFF sólo son aceptadas por las versiones de navegadores más modernos e incluso en estos, con restricciones; las fuentes de tipo SVG y EOT son aceptadas por pocos navegadores.

Es por ellos que solemos ver que en **@font-face** se incluyen varios archivos en **src**:

```
@font-face{  
    font-family:'nombre_fuente';  
    src:  
        url("archivo.eot") format('embedded-opentype'),  
        url("archivo.woff2") format('woff2'),  
        url("archivo.woff") format('woff'),  
        url("archivo.ttf") format('truetype'),  
        url("archivo.svg") format('svg');  
}
```

Luego, simplemente utilizamos el nombre como con cualquier otra fuente:

```
font-family: "nombre_fuente";
```

El CSS3 sigue agregando propiedades para tipografía avanzada pero actualmente, los navegadores aún no reconocen la mayoría de ellas y se duda si alguna vez lo harán o todo quedará en proyectos.

La propiedad **font-feature-settings** se utilizaría en casos muy específicos cuando no haya otro modo de establecer los parámetros de una fuente de tipo OpenType.

La propiedad **font-kerning** define la forma en que cierta fuente espacia sus letras.

Ver más detalles en la [w3.org](https://www.w3.org)

Algo que quizás no se use de modo frecuente, la propiedad **tab-size** nos permite establecer el ancho de las tabulaciones y el valor por defecto es 8.

El valor puede ser el número de espacios:

tab-size: 5;

O cualquier longitud:

tab-size: 2em;

Los textos

Al margen de las fuentes en si mismas, los textos tienen propiedades que podemos controlar como **text-align** que ya habíamos visto y nos permite establecer la alineación del contenido de un elemento.

La propiedad **text-decoration** establece la “decoración” de un texto. Desde la existencia del CSS3, se ha transformado en una propiedad que resume otras tres que se pueden definir de modo separado aunque lo más común es utilizarla sin definir los tres valores, colocando sólo el tipo de línea:

text-decoration: line style color;

La propiedad **text-decoration-line** define el tipo de decoración y puede tener varios valores: **none** (ninguna), **underline** (subrayado), **overline** (subrayado superior), **line-through** (tachado).

La propiedad **text-decoration-style** define el estilo de las líneas y los valores pueden ser **double**, **dotted**, **dashed** o **wavy**.

La propiedad **text-decoration-color** establece el color y si no es definido, será el mismo que el del texto.

La propiedad **text-indent** establece la sangría de la primera línea de un texto dentro de un elemento y los valores que podemos utilizar son similares a los que usamos para los márgenes: cm, mm, in, pt, pc, px, em o ex o un porcentaje.

La propiedad **text-transform** permite que el texto se transforme en mayúsculas, minúsculas o tenga en mayúsculas el primer carácter de cada palabra:

none es el valor por defecto

capitalize la primera letra de cada palabra se transforma en mayúscula

lowercase todas los caracteres se transforman en minúsculas

uppercase todas los caracteres se transforman en mayúsculas

La propiedad **text-shadow** permite agregar una sombra al texto y tiene cuatro parámetros:

text-shadow: posicionX posicionY amplitud color;

posicionX indica el desplazamiento horizontal de la sombra con respecto al texto (positivos hacia la derecha, negativos hacia la izquierda)

posicionY indica el desplazamiento de la sombra con respecto al texto (positivos hacia abajo, negativos hacia arriba)

amplitud es el tamaño de esa sombra, a mayor valor, el efecto de *blur* es mayor

color es obviamente el color de la sombra

Un ejemplo:

`text-shadow: 10px 5px 5px #000;`

Las sombras que se pueden agregar con la propiedad `text-shadow` tienen la particularidad de ser aditivas es decir, se pueden ir acumulando sobre el mismo texto, separando las distintas definiciones con una coma:

`text-shadow: 3px 3px 10px red, -3px -3px 10px yellow;`

La propiedad `text-rendering` es poco difundida y lo que nos permite es controlar la forma en que los navegadores y sistemas operativos muestran los textos cuando se utilizan determinado tipo de fuentes, haciendo hincapié en alguna característica.

Puede tener cuatro valores diferentes:

`auto` es el valor por defecto

`optimizeSpeed` enfatiza la velocidad eliminando ciertas características de la fuente

`optimizeLegibility` enfatiza la legibilidad en desmedro de la velocidad

`geometricPrecision` enfatiza la geometría de la fuente

En la práctica un valor `auto` utiliza `optimizeSpeed` para fuentes pequeñas (menos de 20 pixeles) y `optimizeLegibility` para el resto.

La propiedad puede ser importante con ciertas fuentes como Calibri o Constantia que suelen estar instaladas en Windows y en aplicaciones para dispositivos móviles.

Habíamos visto que en el caso de desbordamientos podíamos usar `word-wrap` para que el texto se distribuyera en varias líneas; sin embargo, hay casos en que esto no es suficiente. La propiedad `text-overflow` nos permite cortar esos textos de otro modo.

Tampoco es una propiedad nueva ya que Microsoft la usó desde siempre pero el CSS3 la hizo universal. No tiene muchos misterios, es sencilla y se aplica a casi cualquier etiqueta siempre y cuando esta posea algunas propiedades extras.

El valor por defecto de esta propiedad es `clip` (el texto se corta y lo que desborda no se muestra). El valor `ellipsis` también corta el texto pero le agrega un carácter elipse al final (tres puntitos); un carácter que también podemos escribir de manera manual utilizando `entities` como `…` (`U+2026`).

Como alternativa, el valor también puede ser una cadena de texto cualquiera:

`text-overflow: ellipsis;`

`text-overflow: clip;`

`text-overflow: "...";`

Para que tenga efecto, la propiedad `overflow` debe ser cualquier valor excepto `visible`, `white-space` debe tener el valor `nowrap` o `pre` y el elemento debe tener definido un ancho con `width`.

La propiedad **white-space** también se utiliza para indicar si las líneas de texto se “cortan” o no y cómo deben ser mostrados los espacios en blanco.

Por defecto, los saltos de línea, los espacios extras (más de uno separando palabras) y las tabulaciones no tienen ningún efecto en una página web. Si queremos agregar espacios extras, debemos utilizar el carácter especial (*nonbreaking space entity*) a menos que establezcamos la propiedad **white-space**.

Podemos usar tres valores distintos:

normal si el texto excede el espacio del contenedor, las líneas se cortan automáticamente y el texto continúa en otra línea (es el valor por defecto)

nowrap el texto se escribe en una sola línea.

pre se utiliza para preservar los saltos de línea y los espacios extras

Como si fuera poco, hay una propiedad más que controla la forma en que se cortan las líneas. Se trata de **word-break** cuyo valor por defecto también es **normal** y admite otros dos: **break-all** y **keep-all** pero su uso es muy eventual porque está restringido a separar textos escritos con caracteres occidentales de textos escritos con caracteres chinos, japoneses o coreanos.

Otras dos propiedades nos permiten establecer la distancia entre palabras y letras.

La propiedad **word-spacing** define una cantidad adicional de espacio entre palabras y el valor debe estar en formato de longitud positivos o negativos.

En forma similar, La propiedad **letter-spacing** define una cantidad adicional de espacio entre caracteres y es muy útil cuando se trata de títulos o fuentes de gran tamaño ya que también acepta valores negativos y eso nos permite “compactar” los textos

En ambos casos, el valor por defecto es **normal**.

La propiedad **line-height** establece la distancia entre las “líneas bases” de un texto y los valores posibles son los mismo que para las otras propiedades: cm, mm, in, pt, pc, px, em o ex o un porcentaje. Por defecto, el valor es **normal**.

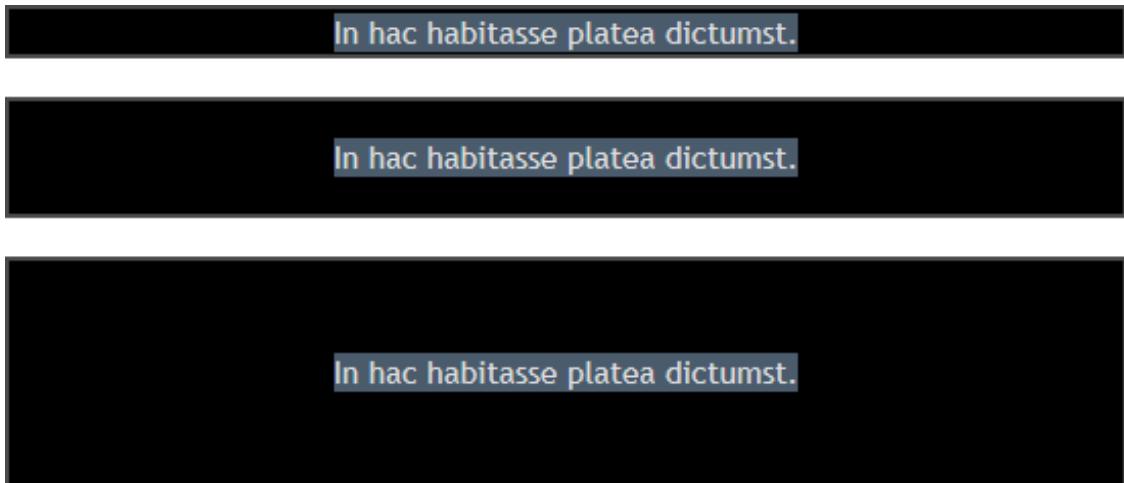
Si el valor es un número, la altura de línea se calcula multiplicando el tamaño de fuente del elemento por ese número.

Podemos decir que **line-height** es la “altura” de la fuente lo que no significa que se modifique la fuente en si misma sino el espacio vertical que ocupa.

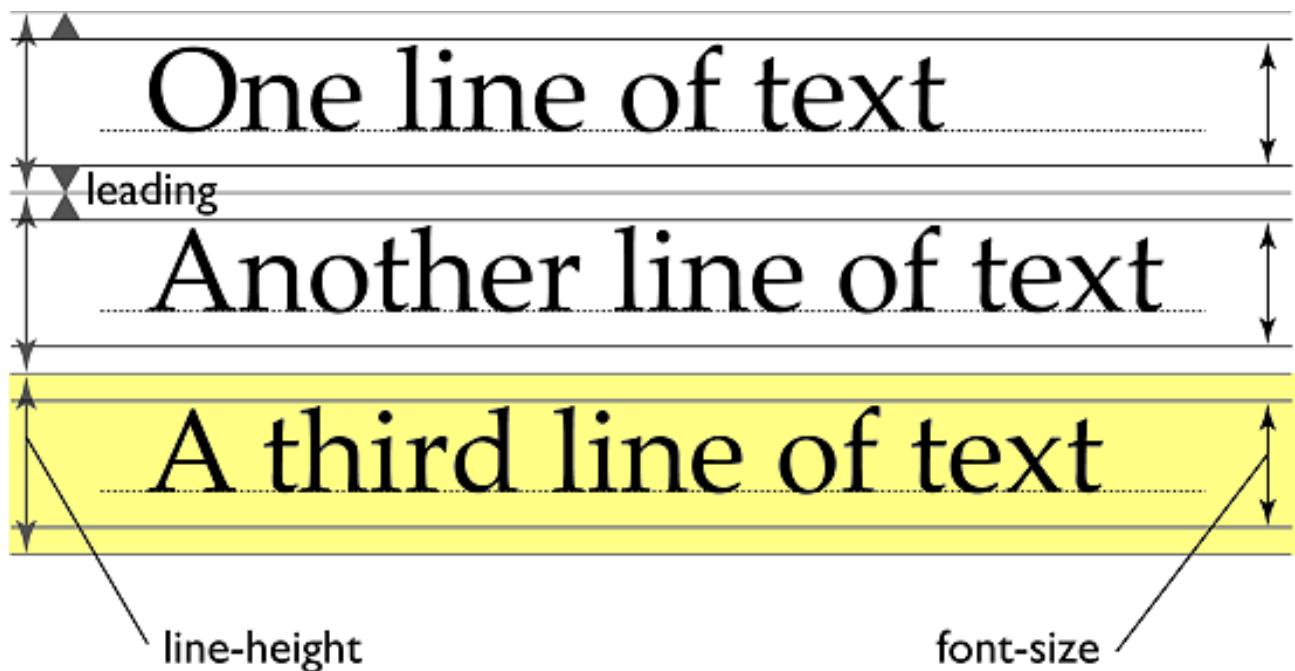
Esta es una propiedad que puede parecer algo confusa ya que, por lo general, es “invisible”; por ejemplo, si la agrego a un elemento *inline* como **** que tenga un color de fondo, no parece tener efecto sobre el texto pero, se “separa” de lo que está arriba y abajo. Es más fácil de ver si coloco esa propiedad en un bloque como un **<div>**:

```
<div style="line-height: 50px;">
    <span style="background-color:#456;">In hac habitasse platea dictumst.</span>
</div>
```

Allí podemos ver que ese rectángulo coloreado y con un borde, al que sólo le vamos cambiando el valor de **line-height** (20, 50, 100), se va haciendo cada vez más alto y el texto, siempre queda centrado verticalmente.

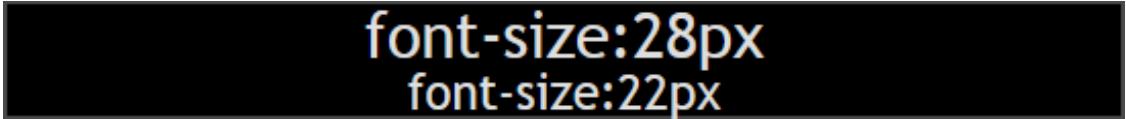


El no tener bien definido ese valor es lo que muchas veces hace que ciertos textos parezcan cortados porque la altura del contenedor es escasa (en ese caso, hay que aumentarlo) o que las líneas aparezcan muy separadas entre si (en ese caso hay que disminuirlo).



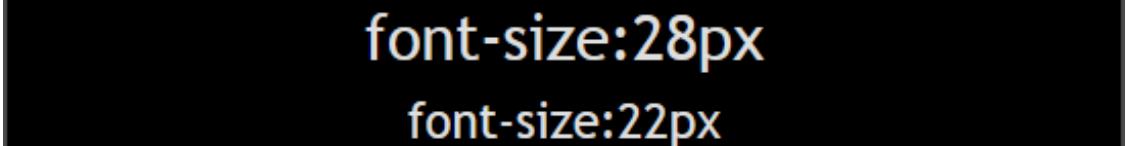
Por defecto, a menos que esté definida globalmente, el valor de **line-height** es 1; eso, significa que la altura de una línea es exactamente igual al tamaño de la fuente.

En este caso, la altura del <div> debería ser de 50 pixeles ya que $28 + 22 = 50$.



font-size:28px
font-size:22px

Usando valores absolutos, la altura se calcula fácilmente, basta multiplicar ese número por el tamaño de la fuente. Si en el ejemplo anterior, **line-height** tuviera el valor 1.5, la altura total sería de: $28 * 1.5 + 22 * 1.5 = 70$ píxeles



font-size:28px
font-size:22px

Otras dos propiedades que se utilizan raramente son **direction** y **unicode-bidi** que se usan de modo combinado.

La primera establece la dirección del texto:

ltr de izquierda a derecha (es el valor por defecto)
rtl de derecha a izquierda

La segunda establece que la forma en que se muestran los textos por defecto debe ser modificada. Su valor por defecto es **normal** y un valor **bidi-override** le dice al navegador que ese texto debe seguir las reglas establecidas por **direction**.

Son propiedades que tienen sentido cuando se trata de páginas web multi-lenguajes pero nada impide usarlas en cualquier otra:

direction: rtl;
unicode-bidi: bidi-override;

También las propiedades de los textos están sufriendo los cambios propuestos por el CSS3 y se van agregando constantemente aunque también se van desechar o sólo son incorporadas por algún navegador.

La propiedad **text-align-last** define la forma en que se muestra la última línea de un texto justificado. Los valores posibles son **auto**, **center**, **left** o **start**, **right** o **end** y **justify**.

La propiedad **writing-mode** define si el texto se muestra de modo horizontal o vertical y la dirección (izquierda o derecha).

El valor por defecto es **horizontal-tb** pero también: **sideways-lr**, **sideways-rl**, **vertical-lr** y **vertical-rl**.

Suele utilizarse junto con **text-orientation** que define la orientación de las líneas de texto pero, sólo tiene efecto cuando **writing-mode** no tiene el valor por defecto. Los valores posibles son: **mixed**, **sideways**, **sideways-left**, **sideways-right**, **upright**, **use-glyph-orientation**.

Por ejemplo:

```
p {  
    text-orientation: sideways-right;  
    writing-mode: vertical-rl;  
}
```

La propiedad **text-emphasis** agrega una línea con una marca (un carácter) sobre cada carácter de la línea inferior excepto espacios y cualquier otro carácter no imprimible.

text-emphasis: style color;

Al igual que otras propiedades, puede dividirse en dos partes. La propiedad **text-emphasis-color** define el color y sigue las mismas reglas que todas.

La propiedad **text-emphasis-style** define el tipo de carácter a mostrar. Por defecto, ese carácter es un punto que puede estar relleno (**filled**) o abierto (**open**):

```
text-emphasis-style: filled;  
text-emphasis-style: open;
```

Los estilos predefinidos son **dot**, **circle**, **double-circle**, **triangle** y **sesame**. Un ejemplo:

text-emphasis-style: open sesame;

Pero, además, ese carácter puede ser cualquier que se nos ocurra colocándolo entre comillas pero, en ese caso no deben usarse las palabras **open** o **filled**:

```
text-emphasis-style: "\2386";  
text-emphasis-style: "+";
```

La propiedad **quotes** indica qué caracteres usar para mostrar las citas (<q>) y por defecto es **none**.

El valor puede ser cualquier par de caracteres; por ejemplo:

```
quotes: "«" "»";
```

Los cursores

Los navegadores utilizan cursores de manera predeterminada, estableciendo cuál será el que se muestra de acuerdo al tipo de etiqueta pero, como casi todo, esos valores pueden ser cambiados utilizando CSS. En este caso, la propiedad que los define es **cursor** cuyo valor es una palabra pre-definida. Los básicos son:

auto es el valor por defecto y el navegador define el tipo de cursor
default muestra la fecha típica de selección

none no muestra ninguno

pointer es el cursor utilizado en los enlaces (una manito)

text es el cursor que indica un área de textos (*i-beam*)

wait se usa para indicar que hay que esperar (un reloj de arena)

help es el cursor que indica algún tipo de ayuda disponible

Se supone que el resto depende del tipo elemento pero nada impide que pueda usarse cualquiera. Por ejemplo:

Los valores que se utilizan para indicar elementos re-dimensionables son: **all-scroll**, **col-resize**, **e-resize**, **ew-resize**, **n-resize**, **ne-resize**, **nesw-resize**, **ns-resize**, **nw-resize**, **nwse-resize**, **row-resize**, **s-resize**, **se-resize**, **sw-resize** y **w-resize**.

Los valores que se utilizan para indicar elementos que pueden ser arrastrados son: **alias**, **copy**, **move**, **no-drop** y **not-allowed**.

Otros valores son: **cell**, **context-menu**, **crosshair**, **progress** y **vertical-text**.

Al entrar en la era del CSS3, a esos cursores se les han sumado algunos otros que por ahora no terminan de aplicarse a todos los navegadores; **grab**, **grabbing**, **zoom-in** y **zoom-out**.

Como si esto fuera poco, la propiedad nos permite utilizar cursores personales, para esto, basta indicar la dirección URL de la imagen que puede ser problemática ya que algunos navegadores aceptan cualquier formato pero otros, sólo archivos en formato CUR o ANI. Por lo tanto, si se opta por esto, lo razonable es colocar siempre un cursor alternativo:

cursor:url("URL_imagen"), auto;

Lo normal es utilizar el valor **auto** como puntero alternativo pero, puede ser cualquier otro.

La propiedad **pointer-events** nos permite controlar si un elemento ignorará el tipo de cursor natural o definido y no reaccionar frente a eventos del ratón. Por defecto, el valor es **auto** y usamos el valor **none** para desactivarlo. Por ejemplo:

este enlace no funcionará

Propiedades de las listas

Las propiedades CSS que controlan la forma en que vemos una lista son pocas pero suelen ser una de las más engorrosas a la hora de controlar el diseño de una página web ya que los navegadores suelen entenderlas de maneras distintas.

La propiedad **list-style** es una forma de establecer todos los valores en una sola instrucción aunque podemos utilizar las propiedades individuales:

list-style: style position image;

La propiedad **list-style-type** establece el aspecto gráfico de las listas y por defecto, su valor es **disc** pero hay una serie de marcadores que podemos emplear aunque no todos los navegadores los aceptan: **circle** y **square** se suelen usar en listas desordenadas (****). En las lista ordenadas (****) las variantes son muchas:

decimal números decimales (1 2 3 4 5 ...)

decimal-leading-zero números decimales formateados (01 02 03 ...)

upper-roman números romanos en mayúscula (I, II, III, IV, V, ...)

lower-roman números romanos en minúscula (i, ii, iii, iv, v,)

lower-alpha letras en minúscula (a, b, c, d, e, ...)

upper-alpha letras en mayúscula (A B C D E ...)

lower-greek alfabeto griego (alpha beta, gamma, ...)

Otros valores se utilizan para idiomas específicos: **cjk-ideographic**, **hebrew**, **armenian**, **georgian**, **hiragana**, **katakana**, **hiragana-iroha**, **katakana-iroha**.

Si no queremos que se muestre ese marcador, el valor a usar es **none**.

Esto puede ser aplicado a cualquier elemento aunque no sea una lista. Para esto, se debe establecer la propiedad **display** como **list-item** y el elemento debe tener un margen izquierdo no menor de 30 puntos.

La propiedad **list-style-position** establece la posición relativa del marcador con respecto al contenido y tiene dos valores posibles, **outside** (el valor por defecto) e **inside**.

Con el primero, el marcador se ubica afuera del texto y si este ocupa varias líneas, los textos se alinean uno debajo del otro. Con el segundo, el marcador se ubica dentro del texto, como si fuera un carácter más.

La propiedad **list-style-image** nos permite definir un marcador personalizado, reemplazándolo por una imagen indicando la dirección URL del archivo:

list-style-image: url(URL_imagen);

Si la imagen no es accesible, es reemplazada por el marcador establecido en la propiedad **list-style-type**.

Además, debe tenerse en cuenta que el margen izquierdo no debe ser cero y para que funcione correctamente, debe ser superior o igual a 30 puntos.

Una característica de las listas es que ese *bullet* o número o marcador comparten el estilo gráfico del contenido es decir, de la etiqueta ; si esta tiene una fuente “grande” o cierto color, esas propiedades también se aplican a la “decoración” por lo tanto, de modo normal, no es posible diferenciarlas.

Propiedades de las tablas

¿Será posible que haya tablas que sin ser tablas sigan siendo tablas? Parece un galimatías pero es lo que ofrecen algunas propiedades recientes del CSS3 que tienen algún uso práctico cuando se trata de alinear cosas.

Una tabla HTML podría ser algo así:

```
<table>
  <caption>esta es una tabla que es tabla</caption>
  <tr>
    <th>Columna 1</th>
    <th>Columna 2</th>
    <th>Columna 3</th>
  </tr>
  <tr>
    <td>Fila 1 Celda 1</td>
    <td>Fila 1 Celda 2</td>
    <td>Fila 1 Celda 3</td>
  </tr>
</table>
```

Esa misma tabla podría ser creada utilizando otras etiquetas como `<div>` y `` estableciendo la propiedad `display` con otros valores:

```
<style>
  #mitabla {display: table;}
  #mitabla .caption {display: table-caption}
  #mitabla .fila {display: table-row;}
  #mitabla .celda {display: table-cell;}
  #mitabla .filatitulo {display: table-row;}
  #mitabla .filatitulo .celda {display: table-cell;}
</style>

<div id="nitabla">
  <div class="caption">una tabla que no es tabla</div>
  <div class="filatitulo">
    <span class="celda">Columna 1</span>
    <span class="celda">Columna 2</span>
    <span class="celda">Columna 3</span>
  </div>
  <div class="fila">
    <span class="celda">Fila 1 Celda 1</span>
    <span class="celda">Fila 1 Celda 2</span>
    <span class="celda">Fila 1 Celda 3</span>
  </div>
</div>
```

En la práctica, casi no veríamos diferencias entre una y otra.

De todos modos, pese a la existencia de las nuevas técnicas, la etiquetas que generan tablas siguen existiendo y tienen algunas propiedades CSS que deben conocerse.

La propiedad **table-layout** define el algoritmo a usar para diseñar la tabla y sólo tiene dos valores posibles: **auto** y **fixed**.

Pero ¿ que significa eso?

El valor **auto** es el valor por defecto y en ese caso, el ancho de las columnas dependen del ancho del contenido de la celda más ancha. Esto, hace que el navegador deba leer todo su contenido para calcular el ancho total antes de mostrarlo.

Por el contrario, un valor **fixed** le indica al navegador que el ancho total y el ancho de las columnas dependen sólo de los valores que se les ha dado y no de su contenido; de ese modo, como no debe calcular nada, la tabla se puede empezar a ver apenas se ha cargado el primer dato.

La propiedad **empty-cells** determina si las celdas vacías de una tabla deberán mostrar sus bordes y colores de fondo aún cuando no hay contenido visible. Tiene dos valores: **show** y **hide**.

La propiedad **border-collapse** se utiliza para definir si los bordes de dos celdas adyacentes se fusionarán.

El valor **separate** separa los bordes y el valor **collapse** los funde en uno.

La propiedad **border-spacing** define la distancia entre los bordes de celdas adyacentes (siempre que **border-collapse** tenga el valor **separate**) y es el equivalente al atributo **cellspacing** que se ha depreciado en HTML.

El valor es una unidad de longitud cualquiera que puede tener uno o dos parámetros (el primero para la separación horizontal y el segundo para el vertical).

Transformaciones

Las transformaciones de un elemento utilizando sólo CSS es tan simple como cualquier otra propiedad:

transform: funcion(parámetros);

Donde los parámetros son funciones varias:

La función **scale(x,y)** aumenta o disminuye el tamaño del elemento; su valor normal es 1. Valores superiores aumentan su tamaño e inferiores lo disminuyen así, 1.5 hará que se vea un 50% más grande y 0.5 un 50% más chico; si sólo se coloca un valor, este se aplica a ambas direcciones, caso contrario, el primero indica el ancho y el segundo el alto.

scaleX(x) y **scaleY(y)** son lo mismo pero sólo afectan al ancho (x) o al alto (y).

Por ejemplo:

```
transform: scale(2);
transform: scale(0.5);
transform: scaleX(2);
transform: scaleY(0.5);
```

La función **rotate(a)** gira un elemento. El valor se expresa en grados, si es positivo gira en el sentido de las agujas del reloj, si es negativo, en el sentido contrario:

transform: rotate(10deg);



La función **skew(a, b)** sesga o inclina elementos y también utiliza ángulos como valor, el primero afecta al eje X (horizontal) y el segundo al eje Y (vertical).

skewX(a) y **skewY(a)** hacen lo mismo pero sólo sobre uno de los ejes.

Por ejemplo:

```
transform: skew(5deg);
transform: skew(5deg, 10deg);
transform: skewX(5deg);
transform: skewY(10deg);
```

La función `translate(x, y)` desplaza el elemento, el primer valor lo hace en el eje horizontal y el segundo en el eje vertical.

`translateX(x)` y `translateY(y)` hacen lo mismo pero sólo sobre uno de los ejes.

```
transform: translate(20px);  
transform: translate(20px, 10px);  
transform: translateX(20px);  
transform: translateY(10px);
```

La función `matrix(a, c, b, d, x, y)` es una combinación de todas las anteriores, por ejemplo:

```
matrix (1, 0, 0, 1, x, y); /* equivale a translate x e y */  
matrix (x, 0, 0, y, 0, 0); /* equivale a scale x e y */  
matrix (1, y, x, 1, 0, 0); /* equivale a skew x e y */
```

Por defecto, un elemento sin transformaciones tendría estos valores:

```
matrix(1, 0, 0, 1, 0px, 0px)
```

Y en este ejemplo:

```
matrix(1.5, 0.1, 0.8, 1.2, -10px, 0px);
```

1.5 es la escala X (*scale*)
0.1 es la inclinación sobre el eje X (*skew*)
0.8 es la inclinación sobre el eje Y (*skew*)
1.2 es la escala Y (*scale*)
-10px es el desplazamiento horizontal (*translate*)
0px es el desplazamiento vertical (*translate*)

Cuando utilizamos las propiedades de transformación, estas se ejecutan tomando como punto de origen el centro del objeto.

Si bien ese punto de origen es el que está definido por defecto, también hay una propiedad que permite cambiarlo llamada `transform-origin` cuyo valor inicial es 50% 50% y para cambiarlo se pueden usar porcentajes, pixeles o palabras como `center`, `left`, `right`, `top` o `bottom`.

El orden es el mismo que en todas las otras propiedades, el primero indica la coordenada horizontal y el segundo la vertical. Así que si pusieramos `0 0` o `left top`, cambiaríamos el punto de origen al ángulo superior izquierdo.

center center o 50% 50% es el centro y el valor por defecto
left top o 0% 0% es el ángulo superior izquierdo
right top o 100% 0 es el ángulo superior derecho
left bottom o 0% 100% es el ángulo inferior izquierdo
right bottom o 100% 100% es el ángulo inferior derecho

También existe una función **perspective()** pero aún no está normalizada y cuyo valor es siempre una longitud que establece la distancia al eje z (si es cero o un valor negativo no hay perspectiva):

```
transform: perspective(valor);
```

El uso de **perspective()** implica que varias funciones tengan una alternativa extra al agregar un tercer eje (el eje z):

```
transform: rotate3d(1,0,0,60deg);
transform: rotateZ(45deg);
```

```
transform: scale3d(1, 2, 1.1);
transform: scaleZ(1.1);
```

```
transform: translate3d(10px, 20px, 100px);
transform: translateZ(100px);
```

```
transform: matrix3d(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3, a4, b4, c4, d4);
```

La propiedad **perspective-origin** es el equivalente 3D a la propiedad **transform-origin** que es exclusiva de las transformaciones 2D:

```
transform: perspective-origin: x, y;
```

La propiedad **transform-style** define si el contenido debe ser ubicado teniendo en cuenta el eje z o no. Los valores posibles son **preserve-3d** y **flat** respectivamente.

Dado la existencia de un tercer eje, la propiedad **backface-visibility** define si la cara trasera de un elemento será visible o no (valores **visible** o **hidden**).

Filtros

La transparencia (o su opuesto, la opacidad) de un elemento puede ser modificada mediante el uso de la propiedad **opacity** donde el valor varía entre 0 (totalmente transparente) y 1 (totalmente opaco):

opacity: valor;

El detalle a tener en cuenta con esta propiedad es que si la aplicamos a un contenedor, todo el contenido es afectado, es decir, si queremos poner un **<div>** con una imagen de fondo y usar **opacity** para hacerla transparente, todo lo que agreguemos dentro de ese **<div>** también será transparente.

La propiedad **filter** nos permite agregar algún tipo de efecto gráfico a cualquier elemento. Esos efectos están definidos por funciones y cada uno de ellos tiene su propia sintaxis.

Lo más sencillo para ver cómo funciona esto es aplicarlos a una imagen y ver el resultado. Por ejemplo, empezamos con cualquier imagen:



La función **blur()** “desenfoca” al objeto y el valor se expresa en pixeles; cuanto más grande sea ese valor, más “desenfocado” estará:

filter: blur(10px);



La función **brightness()** “aclara” al objeto y el valor puede ser un número o un porcentaje. Si es 0% se verá totalmente negro y si es 100% el elemento no cambiará y valores superiores lo mostrarán cada vez más “blanco”:

filter: brightness(0.5);



La función **contrast()** modifica el contraste del elemento. Un valor de 0% lo muestra en tonos de gris; un valor de 100% no lo modifica y valores superiores aumentan el contraste. No es necesario usar porcentajes ya que pueden usarse números:

filter: contrast(300%);



La función **drop-shadow()** es un filtro un poco más complejo ya que requiere varios parámetros que agregan una sombra al elemento.

filter: drop-shadow(x y radio color);

Los valores de x e y establecen la posición horizontal y vertical de la sombra y pueden ser longitudes positivas o negativas; el radio establece la longitud de la sombra, a mayor valor más larga; el color admite cualquier unidad de ese tipo.

filter: drop-shadow(10px 10px 20px red);



La función **grayscale()** convierte al elemento a una escala de grises y el valor define la proporción; el máximo será 100% aunque también pueden usarse números:

filter: grayscale(100%);



La función **hue-rotate()** modifica el tono o matiz de un elemento y el valor debe ser un ángulo. Si es 0deg no hay cambios.

filter: hue-rotate(90deg);



La función **invert()** invierte los colores del elemento y el valor es un porcentaje o un número; Si es 0% no hay cambios y si es 100% la inversión es completa:

filter: invert(100%);



La función **opacity()** es similar a la propiedad **opacity**, agrega cierta cantidad de transparencia a un elemento. Un valor de 0% lo hace completamente transparente y un valor de 100% lo deja sin modificar:

filter: opacity(50%);



La función **saturate()** “satura” los colores y también pueden usarse números o porcentajes:

filter: saturate(200%);



La función **sepia()** convierte el elemento a tonos de sepia. Un valor de 100% lo convierte completamente y un valor de 0% no lo modifica:

filter: sepia(100%);



Todos los filtros pueden ser combinados separándolos con un espacio. Por ejemplo:

filter: brightness(120%) contrast(300%);



Aunque no es un filtro, otra propiedad que causa un efecto en los elementos es **box-shadow** que permite agregarles una sombra a sus bordes. Básicamente tiene esta sintaxis:

box-shadow: x y z d color;

o bien:

box-shadow: x y z d color inset;

Donde los parámetros son estos:

x es la posición horizontal de la sombra (valores positivos o negativos)

y es la posición vertical de la sombra (valores positivos o negativos)

z es el radio de esfumado y es opcional

d es el tamaño de la sombra y también es opcional

color es el color de la sombra y por defecto es negra

inset es una palabra opcional que hace que la sombra se muestre dentro del elemento y no afuera de este

Algunos ejemplos:

box-shadow: 5px 5px 5px 1px rgba(0, 0, 0, 0.5);

box-shadow: 5px 1px yellow inset;

Como otras propiedades, **box-shadow** admite la posibilidad de combinar múltiples sombras separándolas con comas:

box-shadow: 5px 5px red, -5px -5px 1px lightpink;

Transiciones

Una transición es aquello que ocurre entre un momento y otro.

Estamos acostumbrado a ciertos efectos como el colocar el cursor del ratón sobre un enlace y que este resalte de alguna manera; es simple, ese texto tiene un color y cuando ponemos el cursor encima, el color cambia. Ha habido un cambio entre dos estados y esa “transición” es instantánea.

Estas nuevas propiedades, entre otras cosas, nos permiten controlar el tiempo que durará ese cambio entre un estado y otro lo que generará una animación simple porque durante ese proceso, la propiedad o propiedades irán cambiando, yendo de un estado inicial a otro final.

La propiedad básica para conseguir esto se denomina **transition** que es la forma resumida de definir otras cuatro propiedades:

transition: property duration timing-function delay;

La propiedad **transition-property** es la que utilizamos para indicar cuál o cuales son las propiedades que van animarse cuando cambien; si son varias, las sepáramos con comas:

transition-property: color;

transition-property: color, opacity, width;

La propiedad **transition-duration** es el tiempo que durará el efecto y también podemos colocar un solo valor o varios, separados por comas; si hacemos esto último, le estamos diciendo al navegador que cada propiedad indicada por **transition-property** tenga un tiempo de transición distinto.

En todos los casos, el valor está expresado en segundo:

transition-duration: 1s;

transition-duration: 1s, 4s;

Con esas dos ya podemos crear un efecto por ejemplo:

```
p {  
    background-color: white;  
    transition-property: background-color;  
    transition-duration: 2s;  
}  
p:hover {  
    background-color: red;  
}
```

Cuando coloquemos el cursor encima del párrafo, el color de fondo cambiará lentamente y tardará dos segundos en ser rojo.

Casi todas las propiedades pueden sufrir efectos de transición: colores, fondos, bordes, tamaños, posiciones, fuentes, opacidad, sombras, márgenes, etc.

No es necesario enumerar las propiedades a animar, podemos hacerlo con todas a la vez y para eso usamos la palabra **all**:

```
transition: all 1s;
```

O bien:

```
transition-property: all;  
transition-duration: 1s;
```

Cuando queremos anular la transición lo que utilizamos es el valor **none**:

```
transition: none;
```

La propiedad **transition-timing-function** define el modo en que se ejecutará esa transición; la curva de tiempo que se usa para calcular los diferentes estados entre el inicio y el final. Por defecto, tiene el valor **ease** pero pueden usarse otros:

ease genera una animación suave que comienza despacio y luego se va acelerando linear lo hace de modo constante

cubic-bezier() con podemos llegar a controlar esa velocidad de manera exacta aunque compleja

ease-in, **ease-out**, **ease-in-out** son similares a **ease** pero difieren en el modo en que se acelera o desacelera la animación a medida que se llega al tiempo indicado

La propiedad **transition-delay** determina el tiempo que transcurrirá antes de comenzar la transición y su valor inicial es cero.

Un método para combinar transiciones es establecerlas en ambas reglas:

```
p {  
    border-radius: 5px;  
    padding: 10px;  
    transition: padding 2s;  
}  
p:hover {  
    border-radius: 15px;  
    padding: 20px;  
    transition: border-radius 2s;  
}
```

Esto hará que al colocar el cursor encima el párrafo se “redondee” un poco y al quitarlo se “redondee” aún más.

Las transiciones con CSS no están limitadas a efectos *hover* que permutan propiedades de un solo elemento; pueden ser aplicadas de otras formas para reaccionar a clicks o distintos eventos que deben ser ejecutados con JavaScript.

Sin embargo, hay eventos como `:active` que funciona de manera similar a un evento `onclick` ya que se ejecuta justamente cuando pulsamos el botón del ratón y se desactiva cuando lo soltamos.

```
p {  
    font-size: 16px;  
    transition: all 1s;  
}  
p:active {  
    font-size: 100px;  
}
```

¿Cómo funcionará? Dejando pulsado el botón del ratón, el contenido del párrafo irá aumentando hasta el máximo establecido; si soltamos el botón, volverá a su estado natural.

Algo similar puede hacerse en un formulario usando `:focus` que se activará cuando ingresemos un dato:

```
input {  
    border: 2px solid black;  
    width: 100px;  
    transition: width 1s;  
}  
input:focus {  
    width: 300px;  
}
```

Animaciones

Dijimos que las transiciones son la forma de cambiar una o varias propiedades de una etiqueta con un efecto visual tal que ese cambio no se realiza de manera instantánea sino que demora un cierto tiempo y por lo tanto, parece estar animado.

Una animación con CSS es algo similar que requiere de dos partes, primero, establecer los datos de esa animación y luego, agregar las propiedades correspondientes en la etiqueta.

Para crear una animación se utiliza la regla **@keyframes** donde se irán enumerando los pasos intermedios que existirán entre el inicio y el final. Esa regla debe tener un nombre que puede ser cualquiera pero irrepetible:

```
@keyframes NOMBRE {  
    /* aquí irán las reglas de cada etapa */  
}
```

Dentro de **@keyframes** se irán agregando las reglas que se aplicarán en un cierto período de tiempo, ya sea un porcentaje como una palabra clave. Esa lista puede estar en cualquier orden pero como mínimo, debe incluir una regla para el tiempo 0% y 100%:

```
@keyframes NOMBRE {  
    0% {  
        /* las reglas iniciales */  
    }  
    50% {  
        /* las reglas al llegar a la mitad */  
    }  
    100% {  
        /* las reglas finales */  
    }  
}
```

El valor de 0% puede ser reemplazado por la palabra **from** y el de 100% por la palabra **to**:

```
@keyframes NOMBRE {  
    from {  
        /* las reglas iniciales */  
        background-color: black;  
        border: 0px solid gray;  
    }  
    to {  
        /* las reglas finales */  
        background-color: red;  
        border: 15px solid yellow;  
    }  
}
```

Ahora nos falta establecer la propiedad **animation** que es una forma resumida de agregar varias propiedades juntas, algunas de las cuales son obligatorias y otras optativas:

animation: name duration timing delay iteration direction fill play;

La propiedad **animation-name** es el nombre del *keyframe* que asociamos.

La propiedad **animation-duration** es el tiempo que durará esa animación.

La propiedad **animation-timing-function** define el modo en que se ejecutará ese cambio y los valores son los mismos que se usan en **transition-timing-function**.

La propiedad **animation-delay** es opcional y establece el tiempo para que comience la animación y por defecto es cero.

La propiedad **animation-iteration-count** es la cantidad de veces que se ejecutará (por defecto 1) pero podemos usar la palabra **infinite** para que se repita indefinidamente.

La propiedad **animation-direction** indica como se repetirá la animación:

normal lo hará secuencialmente (es el modo por defecto)

alternate lo hará de modo alternado (normalmente en los intervalos impares y inversamente en los pares)

reverse lo hará en sentido inverso

alternate-reverse lo hará de modo inverso en los intervalos impares y normalmente en los pares

La propiedad **animation-fill-mode** indica el estilo del elemento cuando no se está animando o la animación ha terminado. Por defecto es **none** y es el valor que normalmente se utiliza :

forwards especifica que al terminar, el elemento quedará con las propiedades definidas en el último intervalo

backwards especifica que al terminar, el elemento quedará con las propiedades definidas en el primer intervalo

both utilizará ambas

La propiedad **animation-play-state** indica si la animación se ejecuta (**running**) o esta pausada (**paused**).

Algunos ejemplos:

```
#ani1 {animation: animacion1 1s ease 0s normal none infinite running;}  
#ani2 {animation:animacion2 2s linear 0s alternate none infinite running;}  
#ani3 {animation:animacion3 6s linear 0s reverse none 1 running;}
```

Columnas

El uso de columnas para organizar el contenido de un página web es una de las nuevas posibilidades que nos da el CSS3.

Para entender qué significa esto de las columnas, lo mejor es comenzar a entender cómo se ven las cosas de manera normal.

Supongamos que tengo un contenedor (un `<div>`) con un cierto ancho (`width`) y dentro de él, colocamos seis imágenes cuyos anchos son iguales pero su altura varía:

```
#demo {  
    width: 600px;  
}  
  
#demo img {  
    outline: 1px solid Turquoise;  
    width: 200px;  
}
```



Podría establecer la propiedad **vertical-align: top** en las imágenes:



O la propiedad **vertical-align: middle**:



Como miden 200 píxeles de ancho, tres se muestran arriba y las otras tres, debajo; Si quisiera que se vieran filas de cuatro columnas, debería aumentar el ancho del contenedor 200 píxeles más y se acomodarían solas.

Sin otro tipo de propiedad, las imágenes se mostrarán una al lado de la otra, alineadas por su base. Podría alinearlas estableciendo la propiedad **vertical-align** en las etiquetas **** pero, sea como sea y haga lo que haga, las imágenes dejarán espacios en blanco entre ellas ya que todas tienen alturas diferentes y justamente esa es la primera característica sobresaliente de las columnas, que ajustan el contenido de tal modo de no dejar espacios entre ellos.

En el mismo ejemplo podría hacer esto:

```
#demo {  
    column-count: 3;  
    column-gap: 0;  
    width: 600px;  
}  
  
#demo img {  
    outline: 1px solid Turquoise;  
    width: 200px;  
}
```



La propiedad **columns** es una forma simplificada de establecer varias propiedades:

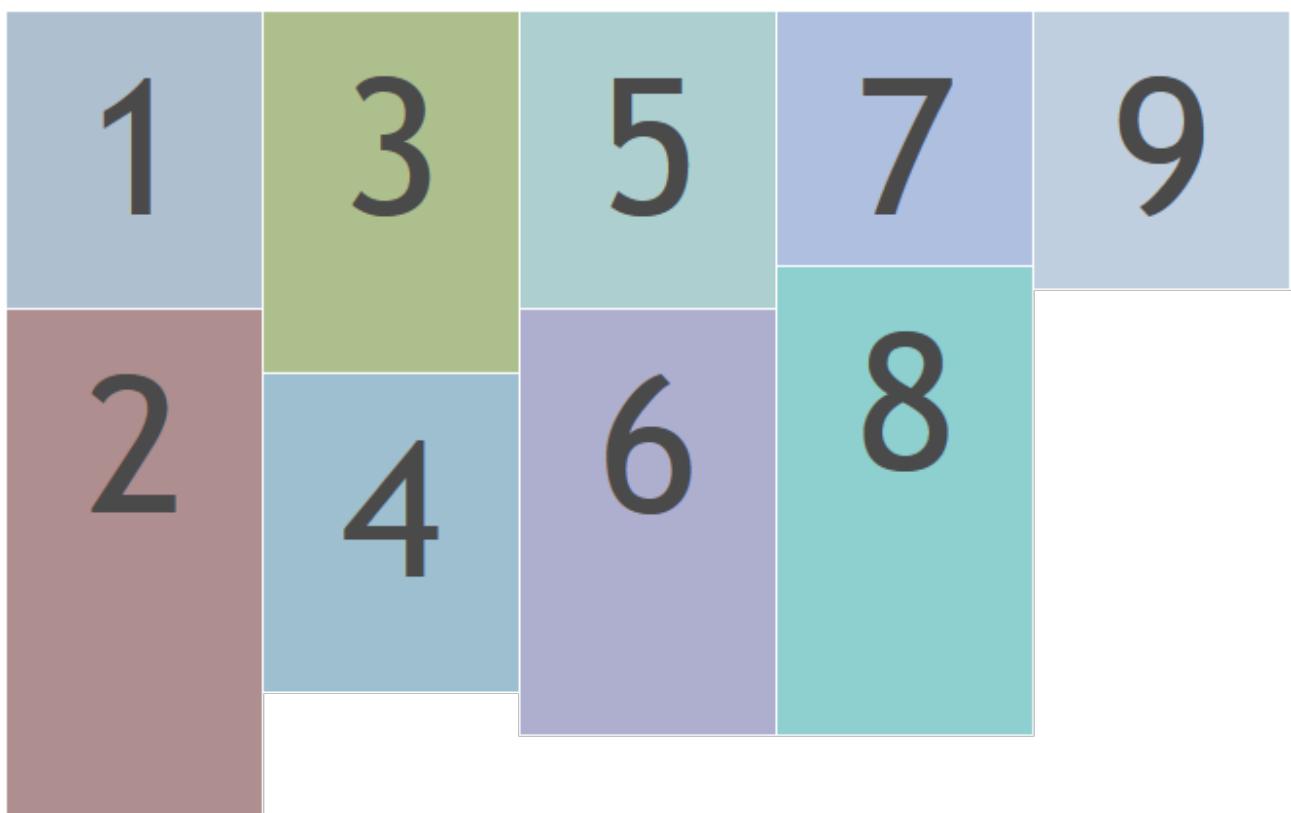
La propiedad **column-count** indica la cantidad de columnas a ser mostradas; si el ancho del contenedor es escaso, se solaparán; si el ancho del contenedor es demasiado grande, quedarán espacios entre ellas

En su lugar de definir la cantidad de columnas podemos usar **column-width** para indicar el ancho de cada una de ellas.

La propiedad **column-gap** es el espacio entre los contenidos (la separación entre ellos); aunque depende del navegador, generalmente es 1em así que podemos poner cero para que la aritmética sea mas sencilla.

La propiedad **column-rule** es opcional y se utiliza para establecer el tipo de borde y tiene la misma sintaxis que la propiedad **border** así que puede ser separada en sus parámetros: **column-rule-color**, **column-rule-style** y **column-rule-width**.

La propiedad **column-fill** es opcional y define como se distribuirá el contenido en las columnas. Un valor de **auto** llenará las columnas secuencialmente; la alternativa es usar **balance** que hará que el contenido se distribuya en equitativamente en todas las columnas.



Flotaciones

Que algo esté alineado (propiedad `text-align`) o que flote (propiedad `float`) no es lo mismo. Puede parecerlo pero es completamente diferente, de allí que muchas veces, nos encontramos con problemas cuando tratamos de ubicar algo en una determinada posición.

El concepto es difícil de asimilar a primera vista pero, debemos partir de esto: todo lo que agregamos a una página web es un rectángulo y básicamente, sólo hay dos tipos de rectángulos; unos, llamados bloques, ocupan el ancho total de donde se pongan y crean un salto de línea, es decir, son “independientes”, se separan de lo que está arriba y de lo que está debajo. El segundo grupo de rectángulos es el llamado *inline*; estos, ocupan el espacio exacto de su contenido.

Un ejemplo con varias etiquetas coloreadas para que se vea cada rectángulo:

```
<div>
    <p>Lorem ipsum dolor sit amet, <strong>consectetuer</strong> adipiscing elit, sed
    <a href="#">diam nonummy</a> nibh euismod tincidunt ut laoreet dolore magna aliquam
    erat volutpat.
        </p>
        <p>Ut wisi enim ad minim veniam, <em>quis nostrud exerci tation ullamcorper
    suscipit lobortis nisl ut aliquip</em> ex ea commodo consequat.
        </p>
</div>
```

Lorem ipsum dolor sit amet, **consectetuer** adipiscing elit, sed **diam nonummy** nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, **quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip** ex ea commodo consequat.

Etiquetas como `<div>`, `<p>` o los encabezados son bloques; etiquetas como ``, ``, `<a>` o `` son *inline*.

Ya vimos que con CSS, cualquier etiqueta puede convertirse de uno a otro tipo, para eso, es que usamos la propiedad `display`. De esta forma, una lista que normalmente es una serie de bloques y ``, la podemos mostrar horizontalmente cambiándoles esa propiedad.

Cuando algo “flota” o sea, cuando le agregamos la propiedad `float` (cuyos valores pueden ser `left`, `right` o `none`), convertimos a ese elemento en un bloque pero con una característica especial: el ancho, a diferencia de un bloque, no es TODO sino sólo su contenido.

Por ejemplo, un encabezado `<h2>` es una etiqueta de bloque, con un color de fondo, normalmente se vería así:

ESTE ES UN ENCABEZADO H2

Podemos cambiarle la propiedad `display` que, por defecto es `block` y ponerle `inline`, veríamos esto:

ESTE ES UN ENCABEZADO H2

Pero lo que quiero es poner eso a la derecha, intento con `text-align:right`:

ESTE ES UN ENCABEZADO H2

No funciona, lo que se alinea es el contenido, así que lo voy a hacer “flotar”:

ESTE ES UN ENCABEZADO H2

¿Qué ocurrió? Se transformó en algo diferente ya no es un bloque en el sentido que ya no ocupa todo el ancho pero tampoco es `inline` porque veamos que pasa si coloco algo arriba y abajo:

```
<p style="color: LightSkyBlue;">Ut wisi enim ad minim veniam ...</p>
<h2 style="float: right;">ESTE ES UN ENCABEZADO H2</h2>
<p style="color: Salmon;">Duis autem vel eum iriure ...</p>
```

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

ESTE ES UN ENCABEZADO H2

El texto en azul no sufre cambios pero el texto rojo que está debajo del objeto que flota, se “acomoda” alrededor de este.

Cambiamos para que flote a la izquierda con:

```
<h2 style="float: left;">ESTE ES UN ENCABEZADO H2</h2>
```

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

ESTE ES UN ENCABEZADO H2

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Pasa lo mismo, todo lo que está debajo de un elemento que flota “sube” y se acomoda alrededor de este.

Es como si a todo elemento que flotara lo quitáramos de la página, le diéramos un tamaño y luego lo empujáramos dentro de esta, ejerciendo presión. Todo lo que estaba allí antes, se moverá y se acomodará como pueda para hacerle lugar.

Si tenemos varios elementos flotantes, estos se ubicarán a derecha o izquierda, según se lo indiquemos y flotarán hasta el borde o hasta encontrarse con otro elemento flotante. Todo lo que está debajo y no flote, se amoldará hasta que, superada la altura del más alto de estos, la página retome su orden natural:

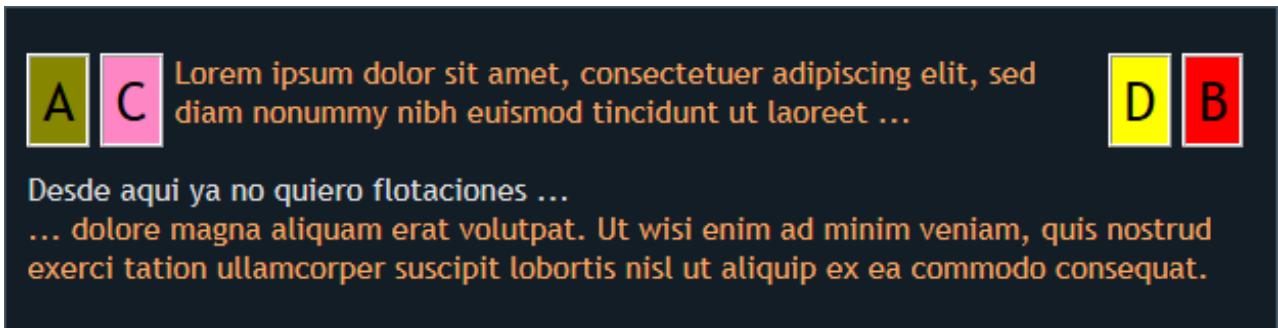
A B C D Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Carece de importancia si flotan hacia un lado u otro, tienen “prioridad” y el resto del contenido se acoda a ellos:

A C Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. D B

Sólo hay algo que permite eliminar las consecuencias de esa flotación, otra propiedad llamada **clear** que tiene cuatro valores posibles: **none**, **left**, **right** y **both**.

Con **left**, le decimos al navegador que a partir de ese lugar, ignore las flotaciones izquierda superiores y lo que sigue, se escriba debajo de forma normal; con **right** hacemos lo mismo para las flotaciones a la derecha y con **both**, le indicamos que lo haga con ambas.



Esa propiedad la agregamos de muchas maneras, por ejemplo en una etiqueta vacía:

```
<div style="clear:both;"></div>
```

Veamos este caso:

```
#contenedor {margin: 0 auto; width:515px;}  
#principal {background-color: #456; float: left; width: 300px;}  
#secundario {background-color: #234; float: right; width: 200px;}  
  
<div id="contenedor">  
  <div id="principal"></div>  
  <div id="secundario"></div>  
</div>
```

Como los navegadores arman la página, leyendo las etiquetas de arriba hacia abajo y de izquierda a derecha, primero se ubica el `<div>` llamado `#principal` que flota a la izquierda y luego el llamado `#secundario` que flota a la derecha.



Como el contenedor mide 515 pixeles de ancho y los contenidos 300 y 200, sobran 15 pixeles que es lo que se separan entre si.

Podría conseguir el mismo resultado haciendo que ambos flotaran a la izquierda y en ese caso, quedarían “pegados” y si quisiéramos separarlos entre si deberíamos usar un margen de 15 pixeles.

Si en lugar de colocarles `float: left` a ambos, le colocamos `float: right`, las cosas cambian, la posición se invierte (el segundo esta en el lugar del primero) pese a que el HTML sigue siendo el mismo:



Esta es la característica principal de la propiedad `float`: deshace el orden natural de las etiquetas, re-ubicándolas.

¿Entonces da lo mismo el orden en que establecemos los distintos elementos de nuestra página porque con CSS podemos ubicarlos en cualquier parte?

Desde el punto de vista visual si pero el orden en que están las etiquetas es fundamental cuando se trata de optimizar un sitio ya que hay algo de sentido común, así como las personas leen de arriba hacia abajo y de izquierda a derecha, lo mismo ocurre con los motores de búsqueda y como estos motores ignoran las propiedades, si el HTML tiene “arriba” el contenido secundario y debajo, el contenido principal lo secundario tendrá prioridad; es lo mismo que ocurriría con un periódico que colocara el titular debajo del artículo y no arriba.

Los motores de búsqueda no entienden de derechas e izquierdas, leen las cosas de manera secuencial, una debajo de la otra y se sabe que no son pacientes así que, simplemente, toman lo primero que encuentran y lo usan para indexar el sitio. Si lo que está primero, es lo secundario, el resultado podría no ser el mejor.

Los elementos agregados a nuestro sitio pueden flotar a la izquierda o a la derecha y no existen términos medios pero ¿para qué hacer que algo flote? El uso más común es para “acomodar” cierto texto alrededor de una imagen:

```
<div>el texto a mostrar</div>
```

```
<div>el texto a mostrar</div>
```



Cras tincidunt auctor metus in interdum. Nam semper varius est, sed pulvinar tellus dignissim vel. Fusce lobortis congue interdum. Morbi lobortis gravida rutrum. Fusce scelerisque fringilla diam, vitae viverra justo rhoncus auctor. Nullam metus purus; sollicitudin sit amet pulvinar ut, euismod at odio. Duis bibendum lacus nec velit posuere pulvinar. Praesent tristique elit a felis tincidunt eu dictum diam ultrices. Donec accumsan posuere accumsan? Maecenas condimentum vulputate ante ac egestas. Suspendisse et enim est, interdum dapibus metus. Nullam venenatis molestie rutrum.

Cras tincidunt auctor metus in interdum. Nam semper varius est, sed pulvinar tellus dignissim vel. Fusce lobortis congue interdum. Morbi lobortis gravida rutrum. Fusce scelerisque fringilla diam, vitae viverra justo rhoncus auctor. Nullam metus purus; sollicitudin sit amet pulvinar ut, euismod at odio. Duis bibendum lacus nec velit posuere pulvinar. Praesent tristique elit a felis tincidunt eu dictum diam ultrices. Donec accumsan posuere accumsan? Maecenas condimentum vulputate ante ac egestas. Suspendisse et enim est, interdum dapibus metus. Nullam venenatis molestie rutrum.



Pero ¿qué pasa cuando queremos centrar esa imagen y ponerle el texto alrededor? No hay respuestas para eso. No está previsto e incluso hay quienes sostienen que es innecesario ya que la legibilidad de los textos se vería seriamente afectada y pese a que eso es una opinión subjetiva, no deja de tener cierta lógica.

VER REFERENCIAS [Colocar texto alrededor de una imagen]

VER REFERENCIAS [Ajustar el tamaño de una imagen al texto]

Si bien son útiles, los elementos flotantes son uno de los que mayores problemas causa. Supongamos que queremos que un elemento contenga a otro:

```
#elContenedor {  
    border: 3px solid #FFFFFF;  
    margin-left: 50px;  
}  
#elContenido {  
    background-color: #8080FF;  
    height: 100px;  
    width: 100px;  
}
```

Definimos algunas reglas de estilo sencillas y el HTML podría ser algo así:

```
<div id="elContenedor">
    <div id="elContenido"></div>
</div>
```



Pero si, le agregamos al contenido la propiedad **float: left**, el resultado será otro:



Esto se debe a que cuando no se especifica un valor para el alto de un elemento, este se calcula a partir del alto de los elementos que contiene (más margenes), pero, si el contenido flota, es lo mismo que “quitarlo” del interior” y el contenedor se queda sin contenido.

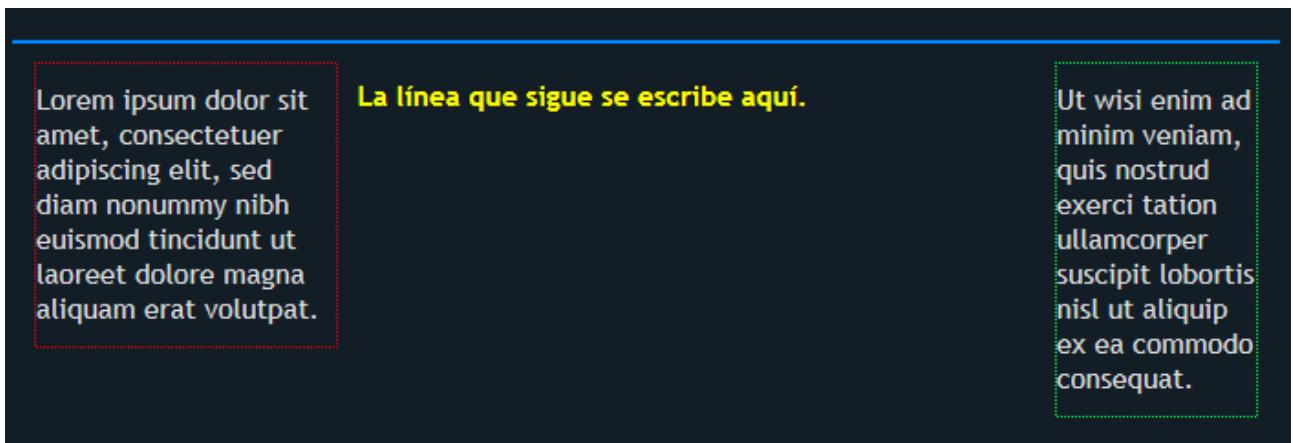
Otro ejemplo:

```
<div id="elContenedor">
    <div id="ladoIzquierdo">
        <p>... textos ... textos ... textos ...</p>
    </div>
    <div id="ladoDerecho">
        <p>... textos ... textos ... textos ...</p>
    </div>
</div>
```

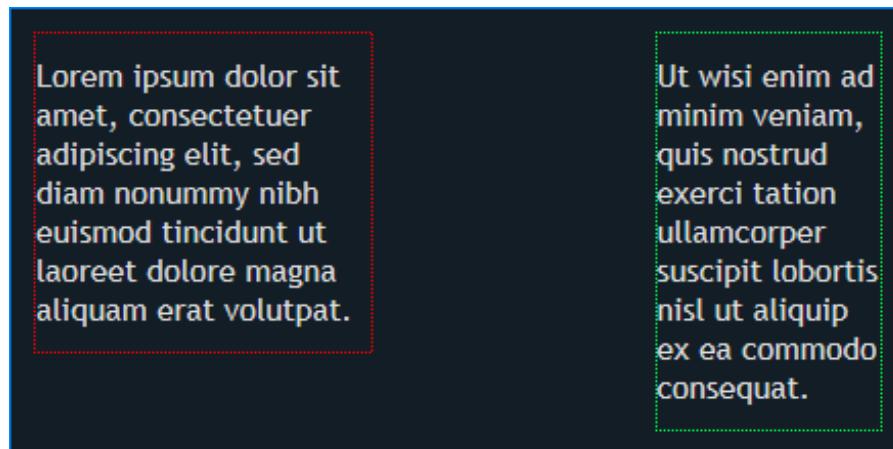
Con estas reglas de estilo:

```
#elContenedor {border: 1px solid #0080FF; margin: auto; text-align: left;}
#ladoIzquierdo {border: 1px dotted #FF0000; float: left; margin: 10px; width: 150px;}
#ladoDerecho {border: 1px dotted #00FF40; float: right; margin: 10px; width: 100px;}
```

El primer problema es que lo que continúa (el texto en amarillo) se “amolda” y aparece donde no debe. Para evitar eso podemos usar la propiedad **clear** que “cancela” la flotación y de esa manera el texto amarillo se verá debajo.



Otra solución es utilizar la propiedad **overflow** para controlar el desbordamiento de los elementos contenidos. Para todos los valores de **overflow** (excepto **visible**) se asume que se ha especificado una dimensión pero, si no es así, es el navegador quien la calcula. Por lo tanto, poniendo la propiedad **overflow: hidden** en el contenedor, este, abarcará al contenido aunque flote:



Basándonos en todo lo anterior podemos diagramar algo más complejo:

```
/* este será el contenedor principal */
#elContenedor {overflow: hidden; margin: auto; width: 400px;}

/* este será el bloque superior y ocupará todo el ancho*/
#elSuperior {height: 50px; margin: 5px 10px; width: 375px; }

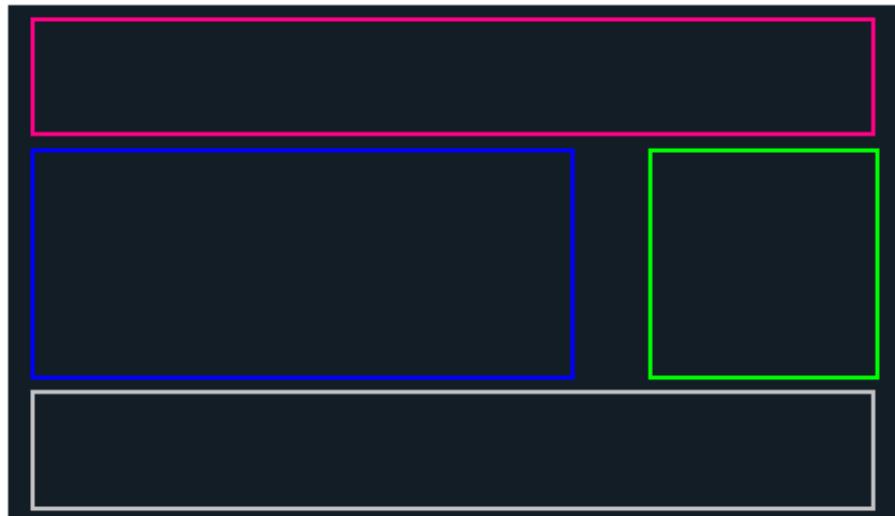
/* el bloque central que flota a la izquierda*/
#elIzquierdo {float: left; height: 100px; margin: 0px 10px; width: 240px; }

/* el bloque central que flota a la derecha*/
#elDerecho {float: right; height: 100px; margin: 0px 10px; width: 100px; }

/* este será el bloque inferior y ocupará todo el ancho */
#elInferior {height: 50px; margin: 5px 10px; width: 375px; }
```

Y este es el código HTML:

```
<div id="elContenedor">
  <div id="elSuperior"></div>
  <div id="elIzquierdo"></div>
  <div id="elDerecho"></div>
  <div style="clear:both;"></div>
  <div id="elInferior"></div>
</div>
```



VER REFERENCIAS [Flotaciones, fondos, problemas, alternativas]

Selectores especiales

Cuando una etiqueta está dentro de otra se dice que la etiqueta contenedora es el elemento padre (*parent*) y la etiqueta contenida es el elemento hijo (*child*). Por eso, siguiendo esta analogía, en CSS se habla de la herencia, es decir, ciertas propiedades definidas en el elemento padre que se “transmiten” al hijo. Por ejemplo:

```
<div style="color: green;">  
  <p> ... un texto ...</p>  
</div>
```

En este caso, el `<div>` es el parent y el párrafo `<p>` es el child. El párrafo se verá verde porque hemos definido que el `<div>` sea de color verde y esta propiedad es heredada.

No todas las propiedades se heredan así que también existen selectores especiales que nos permiten definir estas cosas con mayor exactitud.

Para seleccionar elementos hijos, usamos el símbolo `>` (mayor que) y de ese modo indicamos que cierta propiedad se aplicará sobre ellos. Por ejemplo si tenemos esto:

```
div#ejemplo em {color: blue;}
```

Cualquier etiqueta `` dentro de ese `<div>` se verá de color azul:

En cambio, ya sea usando esto:

```
div#ejemplo p em {color: blue;}
```

o esto:

```
div#ejemplo p > em {color: blue;}
```

sólo serán azules las etiquetas `` que estén dentro de una etiqueta `<p>`.

También podemos seleccionar elementos adyacentes mediante el carácter `+`.

```
p.ejemplo + p {color: yellow;}
```

```
<p class="ejemplo">este es un párrafo cualquiera</p>  
<p>este es el párrafo adyacente al anterior</p>  
<p>este no es el párrafo adyacente</p>
```

Le estamos diciendo que, el párrafo que inmediatamente sigue a uno al que tiene la clase `ejemplo`, sea de color amarillo y todos los demás no.

¿Y que pasa si dos elementos no están adyacentes? El carácter `-` permite seleccionar dos o más elementos separados siempre que todos sean hijos de un mismo parent.

Por ejemplo:

```
em {color: blue;}  
p {color: black;}  
p ~ em {color: red;}
```

En esas reglas decimos que los textos de las etiquetas `` serán de color azul, que los de las etiquetas `<p>` serán de color negro pero que todos los textos en una etiqueta `` que no sean la primera, serán de color rojo:

```
<div>  
  <em>este texto será azul</em>  
  <p>cualquier texto</p>  
  <p>otro texto <em>este texto será azul</em> otro texto</p>  
  <em>este texto será rojo</em>  
  <p>cualquier texto</p>  
  <em>y este texto también será rojo</em>  
</div>
```

Selectores de atributos

Los selectores de atributos hacen justamente eso, permiten que identifiquemos un elemento en función de alguno de sus atributos; se utilizan del mismo modo que cualquier otro y se aplican a etiquetas, clases o **id**.

Por ejemplo esto haría que los párrafos que tuvieran un título se vieran de color amarillo:

```
p[title] {color:yellow;}
```

Lo mismo podría hacerse con cualquier otro atributo (**href src class target alt**).

Pero los selectores permiten mucho más ya que podemos aplicar las reglas de modo más concreto, indicando un valor. De este modo, podemos aplicar una regla de estilo sólo si el atributo tiene un determinado valor:

```
p {color:black;}  
p[title="este es un ejemplo"] {color:yellow;}
```

```
<p>esto se verá de color negro</p>  
<p title="este es un ejemplo">esto se verá amarillo</p>
```

Hay que tener en cuenta que el valor debemos colocarlo entre comillas.

Anteponiéndole el carácter | al signo igual, el atributo debe tener el valor exacto o ese valor continúa con un guión:

```
p {color:black;}  
p[title | ="ejemplo"] {color:yellow;}  
  
<p>esto se verá de color negro</p>  
<p title="ejemplo">esto se verá amarillo</p>  
<p title="otro ejemplo">esto se verá de color negro</p>  
<p title="ejemplo-123">esto se verá amarillo</p>
```

Anteponiendo el carácter ^ seleccionaremos los atributos que comiencen con ese valor:

```
p {color:black;}  
p[title^="ejemplo"] {color:yellow;}  
  
<p>esto se verá de color negro</p>  
<p title="ejemplo de algo">esto se verá amarillo</p>
```

Anteponiendo el carácter ~ seleccionaremos los atributos que contengan determinado valor.

```
p {color:black;}  
p[title~="ejemplo"] {color:yellow;}
```

```
<p>esto se verá de color negro</p>
<p title="un ejemplo sencillo">esto se verá amarillo</p>
```

Anteponiendo el carácter \$ seleccionaremos los atributos que terminen con ese valor:

```
p {color:black;}
p[title$="ejemplo"] {color:yellow;}
```

```
<p>esto se verá de color negro</p>
<p title="un nuevo ejemplo">esto se verá amarillo</p>
```

Anteponiendo el carácter * seleccionaremos los atributos que contengan ese valor total o parcialmente, sin importar su posición:

```
p {color:black;}
p[title*="ejem"] {color:yellow;}
```

```
<p>esto se verá de color negro</p>
<p title="este es un ejemplo de algo">esto se verá amarillo</p>
```

No existen limitaciones en el uso de los atributos y nada nos impide crear uno para satisfacer una necesidad especial, por ejemplo:

```
p[miatributo="cualquiera"] {color: yellow;}
```

```
<p>esto se verá de color negro</p>
<p miatributo="cualquiera">esto se verá amarillo</p>
```

Al igual que los selectores comunes, estos pueden ser combinados:

```
p[title="ejemplo"][class="demo"] {color: yellow;}
```

En ese caso, todos los párrafos `<p>` cuyo atributo `title` sea la palabra `ejemplo` y su atributo `class` sea `demo` serán de color amarillo.

Tampoco hace falta limitarse a verificar una condición ya que puede hacerse con múltiples reglas, por ejemplo:

```
[atributo1^="palabra1"][atributo2$="palabra2"] {color: yellow;}
```

Se verá amarillo el texto de cualquier etiqueta cuyo `atributo1` comience con la `palabra1` y cuyo `atributo2` termine con la `palabra2`.

VER REFERENCIAS [Identificar enlaces por sus atributos]

Pseudo-clases

Las pseudo-clases nos permiten distinguir entre los diferentes tipos de elementos en función de alguna de sus características y su sintaxis es sencilla, sólo se adosa a un selector básico, anteponiéndole el carácter dos puntos (:)

```
selector:pseudoclas {propiedad: valor;}
```

Y como cualquier otra regla, puede ser combinada de múltiples formas:

```
selector.clase:pseudoclas {propiedad: valor;}
```

Empecemos con las mas sencillas que son todas las que podemos utilizar en los enlaces:

```
a {color: red;} /* los enlaces se muestran de color rojo */
```

Ahora agregaremos reglas para los distintos estados usando las pseudo-clases que deberían ser agregadas en este orden:

```
a:link {color: white;} /* son blancos si no los visitamos */  
a:visited {color: green;} /* son verdes si ya los visitamos */  
a:hover {color: yellow;} /* son amarillo si colocamos el puntero del ratón encima */  
a:active {color: blue;} /* son azules cuando están activos */
```

Por lo general, no utilizamos semejante colorinche por lo que, difícilmente nos interese ese orden y sea suficiente establecer dos estados posibles, un enlace “normal” y :hover. Así que podemos simplificar todo de la siguiente manera:

```
a, a:link, a:visited, a:active {color: red;}  
a:hover {color: yellow;}
```

Si bien los ejemplos hacen referencia a enlaces, por ejemplo, :hover podemos aplicarla a cualquier otra etiqueta:

```
#ejemplo:hover {background: black; color: white;}
```

La pseudo-clase :first-child selecciona el primer elemento que se encuentra dentro de otro:

```
div#ejemplo img:first-child {border:2px solid red;}  
  
<div id="ejemplo">  
    
    
</div>
```

Lo que decimos con esta definición es que en un cierto <div>, la primera etiqueta tendrá un borde pero las siguientes no.

Hay variantes. Esto colocará en rojo, todos los textos en negrita de la primera etiqueta `<p>` que estén dentro de cierto `<div>`:

```
div#ejemplo p:first-child strong {color:red}

<div id="ejemplo">
  <p>esto será <strong>rojo</strong> y esto otro <strong>también</strong></p>
  <p>pero ninguno de <strong>estos</strong> se verá <strong>rojo</strong></p>
</div>
```

La pseudo-clase `:last-child` es similar pero hace lo inverso, selecciona el último elemento que se encuentra dentro de otro así que en el mismo ejemplo, la imagen con borde será la segunda:

```
div#ejemplo img:last-child {border:2px solid red;}
```

La pseudo-clase `:first-of-type` selecciona al primer elemento de su tipo dentro de un contenedor:

```
div span {color: black;}
div span:first-of-type {color: red;}

<div>
  <span>esto será rojo</span>
  <span>este texto será negro</span>
  <p>
    <span>y esto también esto será rojo</span>
    <span>este texto será negro</span>
  </p>
</div>
```

La pseudo-clase `:last-of-type` hace lo inverso, selecciona el último elemento de su tipo.

La pseudo-clase `:only-child` selecciona un elemento que es el único dentro de cierto contenedor.

En el ejemplo de las imágenes, ninguna de ellas tendrá borde si usáramos esta regla:

```
div#ejemplo img:only-child {border:2px solid red;}
```

Para que se aplique el borde, sólo debería haber una.

La pseudo-clase `:only-of-type` selecciona cualquier elemento que no tenga adyacente otro de su mismo tipo.

Pero ¿qué podemos hacer para identificar un elemento cualquiera o grupo de elementos dentro de otro sin tener que agregar datos adicionales?

Para eso es que existe la pseudo-clase `:nth-child`.

Por ejemplo, si pusiéramos una serie de párrafos dentro de un `<div>`:

```
<div id="ejemplo">
  <p>el texto 1</p>
  <p>el texto 2</p>
  <p>el texto 3</p>
  <p>el texto 4</p>
</div>
```

Y definimos estos estilos identificando cada párrafo con un número índice:

```
#ejemplo p:nth-child(1) {color: red;}
#ejemplo p:nth-child(2) {color: green;}
#ejemplo p:nth-child(3) {color: yellow;}
#ejemplo p:nth-child(4) {color: blue;}
```

Cada párrafo se verá de un color diferente.

Esto parece demasiado simple pero podemos identificar los elementos con algún tipo de argumento más sofisticado, por ejemplo, esto establecería las propiedades de todos los párrafos impares:

```
#ejemplo p:nth-child(2n+1) { /* propiedades */ }
```

Y esto, todas las propiedades de los párrafos pares:

```
#ejemplo p:nth-child(2n) { /* propiedades */ }
```

nth-child(2n+1) = IMPARES 1 3 5 7	nth-child(2n) = PARES 2 4 6
el primer item de la lista	el primer item de la lista
el segundo item de la lista	el segundo item de la lista
el tercer item de la lista	el tercer item de la lista
el cuarto item de la lista	el cuarto item de la lista
el quinto item de la lista	el quinto item de la lista
el sexto item de la lista	el sexto item de la lista
el septimo item de la lista	el septimo item de la lista

La selección de elementos pares e impares también puede hacerse utilizando las palabras **even** y **odd**:

```
p:nth-child(even) { /* seleccionar items pares */ }
```

```
p:nth-child(odd) { /* seleccionar items impares */ }
```

[VER REFERENCIAS \[Ejemplos de nth-child\]](#)

La aritmética puede ser todo lo compleja que se nos ocurra:

```
/* seleccionar todos los items excepto los primeros cinco */  
ul li:nth-child(n+6) { /* propiedades */ }
```

```
/* seleccionar sólo los primeros tres items */  
ul li:nth-child(-n+3) { /* propiedades */ }
```

La pseudo-clase **:nth-last-child** es similar pero, cuenta los elementos a partir del último; por ejemplo esta regla seleccionaría los últimos tres:

```
ul li:nth-last-child(-n+3) { /* las propiedades */ }
```

Y esto seleccionaría el ante-último:

```
ul li:nth-last-child(2) { /* las propiedades */ }
```

Las pseudo-clases **:nth-of-type** y **:nth-last-of-type** tienen la misma sintaxis pero seleccionan los elementos de cierto tipo.

Existen muchas pseudo-clases asociadas a las distintas etiquetas de los formularios.

La pseudo-clase **:checked** se utiliza en las etiquetas **<input>** de tipo **checkbox** y **radio** y en las etiquetas **<option>**. En todas ellas nos permiten identificar el elemento seleccionado.

```
input[type="radio"]:checked {background:red;}
```

La pseudo-clase **:default** se utiliza para identificar cualquier elemento que se considere el control por defecto dentro de un grupo.

La pseudo-clase **:disabled** identifica las etiquetas **<input>** y **<select>** que estén deshabilitadas (el usuario no puede seleccionarlas). Por el contrario, **:enabled** identifica las etiquetas que estén habilitadas (el usuario puede seleccionarlas).

```
input:disabled {color: grey;}  
input:enabled {color: black;}
```

La pseudo-clase **:optional** se utiliza en las etiquetas **<input>** que no poseen el atributo **required**. Y por el contrario, **:required** identifica las etiquetas cuyo atributo **required** esté habilitado.

La pseudo-clase **:focus** se aplica a los elementos que activamos usando el teclado o el ratón.

Los navegadores muestran distintos estilos gráficos cuando un control posee un dato erróneo y de ese modo, el formulario no es enviado.

Las pseudo-clases `:valid` e `:invalid` se aplican a las etiquetas `<form>` e `<input>` cuyo contenido ingresado se considere válido o inválido en función del tipo de dato que se haya definido.

`:invalid {box-shadow: none;}`

La pseudo-clase `:indeterminate` es similar y se aplica a `<input>` de tipo `checkbox` o `radio` y a la etiqueta `<progress>`.

Las pseudo-clases `:in-range` y `:out-of-range` se aplican a los controles cuyo valor debe estar dentro de cierto rango.

Las pseudo-clases `:read-only` y `:read-write` se aplican a los controles que sólo pueden ser leídos (no modificados) o pueden ser editados por el usuario.

Otras pseudo-clases tienen usos muy específicos como `:fullscreen` que aún no está adoptada por todos los navegadores. Tal como puede inferirse del nombre, identifica cualquier elemento que sea mostrado en el modo pantalla completa.

La pseudo-clase `:lang` identifica elementos basándose en el lenguaje determinado por la etiqueta `<meta>` o el protocolo HTTP.

La pseudo-clase `:target` es algo más compleja de comprender. Selecciona a un elemento único cuyo id es igual al valor del atributo `href` de un enlace de tipo “anchor” (enlaces a alguna parte de la misma página).

Supongamos una regla así:

`:target {background-color: red;}`

Aplicada a un ejemplo sencillo:

```
<p><a href="#ejemplo1">mostrar ejemplo 1</a></p>
<p><a href="#ejemplo2">mostrar ejemplo 2</a></p>

<p id="ejemplo1">este es el párrafo 1</p>
<p id="ejemplo2">este es el párrafo 2</p>
```

Al hacer *click* en el primer enlace (`href="#ejemplo1"`), el párrafo cuyo id es `ejemplo1` cambiará de color de fondo y al hacer *click* en el segundo, resaltará el segundo pero siempre, sólo uno de ellos.

Como cualquier otro selector, podemos ser más específicos:

`h3:target {background-color: red;}`

`div#cualquiera p:target {background-color: red;}`

Una pseudo-clase muy interesante es `:empty` que selecciona un elemento siempre que este no contenga elementos internos (hijos) ni textos de ninguna clase.

La pseudo-clase `:not` es otra de las interesantes porque nos permite excluir ciertos elementos.

Por lo general, cuando queremos que las etiquetas de cierto tipo se vean de manera especial, les agregamos una clase. Usando `:not` podemos hacer lo mismo, definir una clase general y luego, hacer que no se aplique en ciertos casos. Un ejemplo:

```
.ejemplo {color: black;}  
.ejemplo:hover:not(.esteno) {  
    color: red;  
}  
  
<p><a class="ejemplo" href="#">un enlace normal</a></p>  
<p><a class="ejemplo esteno" href="#">este enlace NO será rojo</a></p>  
<p><a class="ejemplo" href="#">otro enlace normal</a></p>
```

Al colocar el cursor encima de cada enlace, el segundo no se verá de color rojo.

Otro ejemplo donde establecemos que todas las imágenes tendrán un borde excepto aquellas cuya clase sea `.sinborde`:

```
img {  
    border: 5px solid red;  
}  
img.sinborde {  
    border: none;  
}
```

Esto funcionará perfectamente pero usando `:not` podemos simplificar el CSS ya que podemos combinar ambas reglas en una sola:

```
img:not(.sinborde) {  
    border: 5px solid red;  
}
```

VER REFERENCIAS [Eventos click y CSS]

La pseudo-clase `:any` se puede utilizar para reducir el número de reglas ya que reducen las declaraciones de manera sustantiva, creando grupos de selectores.

Aunque aún no está estandarizada, puede usarse con prefijos (`-moz` y `-webkit`) sin problemas. La sintaxis básica es:

`:any(lista de selectores);`

O bien:

`:-moz-any(lista de selectores);`
`:-webkit-any(lista de selectores);`

Donde la lista de selectores se separan con comas.

Por ejemplo, supongamos que queremos que una serie etiquetas tuviera un cierto color de fondo; podríamos escribir:

```
div, p, li {  
    background-color: grey;  
}
```

Podríamos escribirlo de este otro modo:

```
:any(div, p, li) {  
    background-color: grey;  
}
```

La pseudo-clase `:scope` también es experimental aunque no requiere prefijos. Si bien su uso es simple de entender, requiere alguna explicación previa.

La etiqueta `<style>` tiene dos atributos optativos: `type` (que suele ser `text/css`) y `media` que indica el tipo de dispositivo donde se aplicará. Esto es lo usual pero, hay un tercer atributo experimental llamado `scoped`.

Si agregamos una etiqueta `<style scoped>` dentro de alguna etiqueta de nuestra página, las reglas de estilo definidas allí se aplicarán sólo al elemento padre.

Veamos un ejemplo. Tenemos varias etiquetas `<p>` con textos que, por defecto, se verán de color negro; dentro de ellas hay etiquetas `` que también se verán de color negro pero queremos que solo las del segundo párrafo se vean de color azul.

Podríamos agregarles un estilo *inline* a cada una de ellas:

```
<p>un texto cualquiera <span>de color negro</span></p>  
<p>  
    este es el párrafo que modificamos  
    <span style="color: blue;">color azul</span>  
    textos textos textos  
    <span style="color: blue;">color azul</span>  
</p>  
<p>otro texto cualquiera <span>de color negro</span></p>
```

Pero, también podríamos usar `<style scoped>` y lo haríamos modificando el segundo párrafo de este modo:

```
<p>  
    <style scoped>p span {color: blue;}</style>  
    este es el párrafo que modificamos  
    <span>color azul</span>  
    textos textos textos  
    <span>color azul</span>  
</p>
```

Esa regla que hemos agregado, sobrescribe los estilos que hayamos declarado previamente y sólo se aplicará a ese párrafo y ninguno de los otros se verá afectado.

Claro, también podríamos haber colocado una clase y definirla al inicio pero este nuevo método sigue siendo aceptable porque si se trata de un elemento eventual, que sólo usamos en un lugar, los estilos globales permaneces “limpios”.

La pseudoclase `:scope` es una derivación de lo anterior pero más limitada ya que no nos permite definir reglas diversas sino sólo una que afecta a todos los elementos del contenedor.

Por ejemplo:

```
<style scoped>
  :scope {color: red;}
</style>
```

Hará que todos los textos del segundo párrafo se vean de color rojo, estén dentro de un `` o no.

Pseudo-elementos

Los pseudo-elementos seleccionan alguna parte de un elemento y su sintaxis es similar a las pseudo-clases aunque a veces, en lugar de usar un carácter dos puntos se utilizan dos.

```
selector::pseudoelemento {propiedad: valor;}
```

Los más sencillos de comprender son `::first-letter` y `::first-line` que permiten identificar la primera letra de un texto o la primera línea y establecer propiedades particulares para cada una de ellas.

El pseudo-elemento `::first-line` puede usarse en cualquier elemento de bloque como `<p>`, `<h1>`, etc o que posean la propiedad `display` con valores como `block`, `inline-block`, `table-cell` o `table-caption`. Por ejemplo:

```
p::first-line {  
    font-variant: small-caps;  
    font-weight: bold;  
}
```

En este otro ejemplo, el pseudo-elemento `::first-letter` se emplea para que la letra inicial sea tres veces mayor que el tamaño normal de fuente:

```
p::first-letter {  
    font-size: 300%;  
}
```

Suspendisse lacinia erat ut ipsum tristique at pretium nunc bibendum! Mauris non neque a odio ultrices volutpat.

Suspendisse lacinia erat ut ipsum tristique at pretium nunc bibendum! Mauris non neque a odio ultrices volutpat.

En ambos casos, no todas las propiedades tienen efecto, sólo las que afectan a las fuentes (`font`), al fondo (`background`), al color, algunas otras como `text-decoration`, `text-shadow`, `text-transform`, `word-spacing`, `letter-spacing`, `line-height` y `vertical-align`.

El pseudo-elemento `::first-letter` admite algunas más como `margin`, `padding`, `border`, `box-shadow` y `float`.

Como en cualquier otra regla, podemos combinar ambas cosas y, en ese caso, hay que saber que “quien mande” será `::first-letter`, sin importar el orden en que lo hayamos escrito:

```
Mauris non neque a odio ultrices volutpat. Aliquam orci lectus; consequat  
at ultrices ac, luctus sed erat. Duis in pretium orci. Phasellus metus leo,  
hendrerit at volutpat.
```

El pseudo-elemento `::selection` nos permite establecer propiedades personales cuando seleccionamos algún texto aunque sólo admite algunas propiedades como `background-color`, `color`, `cursor`, `outline`, `text-decoration`, `text-emphasis-color` y `text-shadow`.

```
::selection {  
    background-color: darkred; color: yellow;  
}
```

Los pseudo-elementos `::after` y `::before` nos permiten agregar contenido a una etiqueta; podríamos decir que el resultado es parecido a que a la etiqueta se le adosaran otras dos, una antes (`before`) y otra después (`after`).

No tienen demasiadas restricciones y se pueden aplicar a cualquier etiqueta. La sintaxis básica es:

```
selector::after {propiedad: valor;}  
selector::before {propiedad: valor;}
```

Si bien podemos usar cualquier propiedad, hay una especial que es la que nos permite agregar contenido que puede ser tanto un texto como la dirección URL de una imagen; esa propiedad es `content`:

```
selector::after {  
    content: url(URL_imagen);  
}  
selector::before {  
    content: "un texto cualquiera";  
}
```

Imaginemos un HTML con un texto con un enlace común:

```
<div>  
    Etiam venenatis ultrices hendrerit.  
    <a href="#" class="demo">Nunc lectus enim</a>,  
        faucibus sit amet laoreet et, pulvinar sit amet sem.  
</div>
```

Estableceremos las reglas para adosarle una imagen delante del enlace y un texto detrás:

```

.a.demo::before {
    content: url(URL_imagen);
    padding-right:3px;
    vertical-align: middle;
}
.a.demo::after {
    color: #FF0;
    content: " [más información]";
    font-size: 80%;
    font-family: monospace;
}

```

Etiam venenatis ultrices hendrerit.  Nunc lectus enim [más información], faucibus sit amet laoreet et, pulvinar sit amet sem.

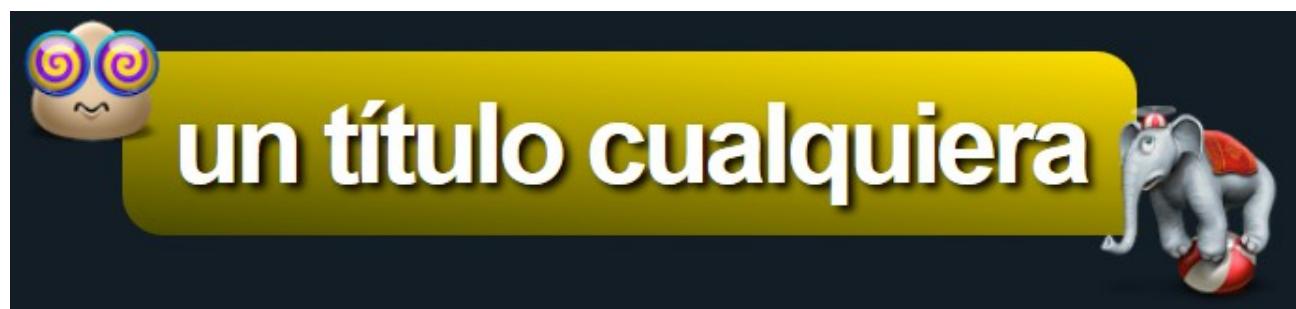
De este modo, `::before` es lo que está antes y `::after` es lo que está después pero ¿antes y después de qué? Simplemente de lo que está antes y después de una etiqueta pero no se trata de las etiquetas o contenidos adyacentes sino que podríamos decir que es como dividir una etiqueta en tres partes, `::before` es eso que está en el extremo izquierdo de una etiqueta y `::after` es lo que está en el extremo derecho de esa etiqueta.

Si establecemos la propiedad `position: relative` a la etiqueta que queremos modificar, tendremos la posibilidad de posicionar de modo absoluto esos contenidos extras.

```

.titulodemo {
    position: relative;
}
.titulodemo:after, .titulodemo:before {
    position: absolute;
}
.titulodemo:after {
    content: url(imagen_izquierda);
    right: valorpx;
    top: valorpx;
}
.titulodemo:before {
    content: url(imagen_derecha);
    left: valorpx;
    top: valorpx;
}

```



Hasta acá hemos usado ::after y ::before con la propiedad **content** contenido un texto o una imagen pero, en realidad no es necesario que tenga contenido alguno, podemos colocar como valor una cadena de texto vacío.

Supongamos que tenemos un <div> con una imagen:

```
<div class="ejemplo">
  
</div>
```

Agregaremos reglas para que la imagen se “entinte” con un color de fondo y se vea natural al colocar el cursor encima:

```
.ejemplo {
  cursor: pointer;
  display: inline-block;
  position: relative;
}
.ejemplo :before {
  background: rgba(0,255,255, 0.5);
  bottom: 0;
  content: "";
  display: block;
  left: 0;
  position: absolute;
  right: 0;
  top: 0;
  transition: background .3s linear;
}
.ejemplo:hover:before {
  background: none;
}
```

Ya vimos que **content** puede tener textos, imágenes o estar vacíos pero hay otros contenidos posibles.

counter() se utiliza para crear un contador combinado con la propiedad **counter-increment** que en su forma más simple, tiene como valor un nombre que hayamos elegido. Eventualmente, usaremos la propiedad **counter-reset** para poner a cero el contador.

counter-increment: indice;

Veamos un ejemplo; tenemos una serie de párrafos y usaremos ::before para adosarles un número delante de cada uno, un número que será el número de orden de ese párrafo:

```
<p class="demo">el texto del párrafo 1</p>
<p class="demo">el texto del párrafo 2</p>
<p class="demo">el texto del párrafo 3</p>
```

```
p.demo {  
    counter-increment: indice;  
}  
p.demo:before {  
    content: counter(indice);  
    color: red;  
    font-size: 150%;  
    padding-right: 1em;  
    vertical-align: middle;  
}
```

Nada impide que usemos **counter()** y textos combinados:

```
content: counter(indice) " -> ";
```

open-quote y **close-quote** adosan una comilla abierta o cerrada:

```
p.demo:before {content: open-quote;}  
p.demo:after {content: close-quote;}
```

La función **attr()** devuelve el valor del atributo de un elemento y podemos usarla con la propiedad **content** pero también con cualquier otra propiedad.

Supongamos que tenemos una etiqueta que muestra una cantidad que varía en función de algún tipo de acción del usuario:

```
<p id="demo" cantidad="5">la cantidad es</p>
```

Podríamos “leer” ese atributo personal que hemos llamado cantidad y usar **::after** para agregarlo atrás de la etiqueta:

```
#demo::after {  
    content: attr(cantidad);  
}
```

A veces, en **content** es necesario usar caracteres “raros” o “especiales” como letras acentuadas y estos no se ven bien o aparece un signo de interrogación. Por lo general, esto se resuelve si la codificación de caracteres es correcta pero lo lógico no siempre funciona y, en esos casos, debemos escribirlos en un formato especial, las llamadas cadenas de escape que no son otra cosa que una barra invertida seguida por un número que, de manera genérica es el código Unicode en formato hexadecimal (cuatro dígitos):

En esta [página](#) hay un listado muy completo de los códigos que pueden ser utilizados.

Por ejemplo:

```
content: "\00e1 \00e9 \00ed \00f3 \00fa"; /* escribirá á é í ó ú */
```

Navegadores y prefijos

Tendemos a creer que la tecnología web está estandarizada, que todos los dispositivos “entienden” los mismos lenguajes y que todos los usuarios verán lo mismo cuando naveguen pero eso no es así en absoluto.

El CSS no es la excepción.

La w3.org es la encargada de establecer esos estándares pero quienes desarrollan páginas web se topan con tres inconvenientes. Por un lado, los navegadores van incorporando esas recomendaciones lentamente; además, muchas de esas reglas o propiedades se encuentran en proceso de estudio y para colmo, los navegadores proponen el uso de otras o insisten en crear sus propias reglas.

Es por eso que al no haber un estándar que respeten todos en un 100%, a veces se requiere que ciertas propiedades se definan anteponiéndoles un prefijo.

- moz- para navegadores basados en Gecko como Firefox
- webkit- para navegadores basados en Webkit, como Chrome, Safari, Android e iOS
- o- para Opera (aunque este navegador ha comenzado a usar el motor Webkit)
- ms- para Internet Explorer
- khtml- para Konqueror

Por ejemplo, la propiedad **appearance** es experimental sin embargo, podría usarse en algunos navegadores de este modo:

```
-moz-appearance: valor;  
-webkit-appearance: valor;
```

Estos prefijos ([vendor prefixes](#)) se aplican exclusivamente a cada navegador por lo tanto si queremos utilizar alguna de esas propiedades, deberemos agregarlos todos

Si bien es cierto que esas propiedades pueden ser utilizadas y en la mayoría de los casos funcionarán correctamente, es bueno preguntarse si deberíamos hacerlo.

Lo primero que deberíamos definir es si es necesaria ya que quizás existe alguna alternativa estandarizada y podemos prescindir de ella.

Si no existe alternativa, deberíamos listar todos los prefijos:

```
.ejemplo {  
    -ms-propiedad: valor;  
    -moz-propiedad: valor;  
    -o-propiedad: valor;  
    -webkit-propiedad: valor;  
    propiedad: valor;  
}
```

El orden en que las listamos carece de importancia pero siempre hay que tomar la precaución de agregar la propiedad sin prefijo al final de la regla.

Es muy importante saber que la lista de propiedades que necesitan un prefijo cambia constantemente. Si miramos el código fuente de muchos sitios, veremos que están agregadas algunas que son innecesarias ya que los navegadores ya aceptan las propiedades estándar (*border-radius*, *linear-gradient()*, *box-shadow*, etc).

Por último, no es recomendable utilizar prefijos para soportar navegadores obsoletos.

at-rule

¿A que se llama *at-rule*? ¿Qué significa esto? En CSS son reglas especiales que comienzan con el carácter arroba (en inglés *at*) y ya hemos visto algunas de ellas como **@import**, **@font-face** y **@keyframes** pero hay otras que podemos aplicar sin problemas y un sinfín que aún están en proceso de experimentación.

@charset es una de las más antiguas y permite definir la codificación de caracteres a utilizar en una hoja de estilo por lo tanto, debe ser la primera regla incluida y no puede estar precedida por ningún otro carácter, incluyendo espacios o tabulaciones.

```
@charset "UTF-8";
@charset "iso-8859-15";
```

@page se usa para establecer las reglas de estilo a utilizar cuando se imprime la página aunque sólo pueden definirse ciertas propiedades básicas como los márgenes y una serie de propiedades y pseudo-clases específicas como **orphans**, **page-break-after**, **page-break-before**, **page-break-inside**, **widows**, **:first**, **:left**, **:right**, etc.

```
@page {
    margin: 2cm;
    orphans: 4;
    widows: 2;
    h1 {
        page-break-before : right;
    }
}
@page :first {
    margin-top: 10cm;
}
```

@namespace tiene un uso muy específico y sólo se aplica cuando se incluye contenido SVG o XML que debe ser interpretado.

```
@namespace url(http://www.w3.org/1999/xhtml);
@namespace svg url(http://www.w3.org/2000/svg);
```

@viewport permite agregar reglas específicas para que sean utilizadas por distintos dispositivos como teléfonos móviles, etc. Lamentablemente, aún no está estandarizada así que deberíamos ser prudentes al utilizarla.

Algunas de las propiedades que pueden incluirse son las típicas que definen las dimensiones (**width**, **min-width**, **max-width**, **height**, **min-height**, **max-height**) pero otras son exclusivas de este tipo de regla (**zoom**, **min-zoom**, **max-zoom**, **user-zoom**, **orientation**).

```
@viewport {  
    min-zoom: 0.5;  
    max-zoom: 0.9;  
    min-width: 640px;  
    max-width: 800px;  
    orientation: landscape;  
    zoom: 0.75;  
}
```

@media nos permite especificar las propiedades que tendrá nuestro sitio según sea el medio en el cual se mostrará: una pantalla, un papel impreso o incluso algunos navegadores de voz. De ese modo es posible diferenciar lo que se muestre en uno o en otro.

La sintaxis genérica sera esta:

```
@media dispositivo {  
    /* reglas de estilo para ese dispositivo */  
}
```

Los dispositivos se identifican mediante palabras claves:

all se aplicará a todos
print se aplicará a las impresoras
screen se aplicará a las pantallas
speech se aplicará a los sintetizadores de voz

Versiones previas del CSS definían muchos otros dispositivos pero todos ellos están depreciados y deben evitarse (*aural, braille, embossed, handheld, projection, tty, tv*).

Este tipo de regla tiene la particularidad de permitir el uso de operadores lógicos (*and or not*) de tal forma de crear condiciones más complejas. Esas condiciones están basadas en alguna característica ([media queries](#)):

height evalúan el alto de la pantalla
width evalúan el ancho de la pantalla
aspect-ratio se refiere a la relación entre el ancho y el alto de la pantalla
orientation se refiere a la orientación de la pantalla (horizontal o vertical)
resolution se refiere a la densidad de los pixeles de la pantalla
color se refiere al número de bits por color
color-index indica el uso de una tabla de colores
monochrome indica la cantidad de bits por pixeles de un dispositivo blanco y negro

Muchas de ellas admiten el uso de los prefijos **max-** y **min-** así que también tenemos: **max-height, min-height, max-width, min-width, max-aspect-ratio, min-aspect-ratio, max-color, min-color, max-resolution, min-resolution, max-color-index, min-color-index, max-monochrome, min-monochrome**.

También podemos utilizar **device-height, device-width, max-device-height, min-device-height, max-device-width y min-device-width**.

Algunos ejemplos:

```
@media (aspect-ratio: 16/9) {  
    /* propiedades para pantallas de tipo widescreen */  
}  
@media (orientation:portrait) {  
    /* propiedades para pantallas más altas que anchas */  
}  
@media (orientation:landscape) {  
    /* propiedades para pantallas más anchas que altas */  
}  
@media (color) {  
    /* propiedades para monitores color */  
}  
@media (min-color-index: 256) {  
    /* propiedades para monitores con 256 colores o más */  
}
```

El uso más común de **@media** es crear páginas que se adapten en distintos dispositivos, ya sea que tengan pantallas pequeñas o grandes y esto puede hacerse combinando las condiciones.

Lógicamente, la amplia variedad de dispositivos y resoluciones hace que sea imposible resolverlas todas ((PC de escritorio, tabletas, laptops, teléfonos) y como no hay estándares para esto, sólo podemos establecer algunas de las más comunes: 320, 480, 600, 800, 1024, 1200.

Por ejemplo, esto se podría aplicar a resoluciones mínima de 1200 pixeles de ancho:

```
@media (min-width: 1200px) {  
    /* propiedades */  
}
```

Esto aplicaría reglas de estilo para resoluciones entre 480 y 800 pixeles de ancho:

```
@media screen and (max-width:800px) and (min-width: 480px) {  
    /* propiedades */  
}
```

Variables y propiedades personales

Una alternativa que aún es experimental pero que ya es aplicable en Chrome, Firefox, Opera y Android es la posibilidad de crear variables que luego podemos utilizar en nuestra hoja de estilo y de ese modo definir valores que repetiremos una y otra vez.

Esas propiedades personales deben ser escritas anteponiéndole dos guiones (--) seguidas de una palabra cualquiera que luego usaremos con la función `var()`.

Una forma práctica de implementar esto es utilizar una pseudo-clase que no habíamos enumerado y se llama `:root` que simplemente, representa la raíz (*root*) del documento o sea, el elemento `<html>`.

Un ejemplo:

```
:root {  
    --colorfondo: lightgray;  
    --colortextos: white;  
    --bordes: 2px dotted black;  
}
```

Allí, hemos definido dos variables personales cuyo valor es un color y otra cuyo valor es un estilo de borde.

Ahora, podríamos aplicar esas variables a las reglas comunes; por ejemplo:

```
body {  
    background-color: var(--colorfondo);  
}  
p {  
    color: var(--colortextos);  
    border: var(--bordes);  
}
```

La página tendrá el color de fondo definido con *colorfondo* y los textos de los párrafos tendrán el color definido con *colortextos* y un borde tal como fue definido con la variable *bordes*.

El uso de `:root` es opcional, si quisiéramos, podríamos declarar esas propiedades personales en un selector específico o una clase y, en ese caso, esas variables sólo tendrían efecto en la reglas de ese selector.

```
p {  
    --mi-color: red;  
}  
p.ejemplo {  
    color: var(--mi-color, red);  
}
```

En ese ejemplo estamos modificando levemente la sintaxis de `var()`, agregando un segundo parámetro opcional, separado con una coma. Esta es la forma de evitar que, si el primer dato (el nombre de la variable personal o su valor) no es válido, se utilizará el segundo dato que siempre debería ser un valor estándar.

Quizás alguno se preguntará si estas cosas tienen algún sentido a la hora de diseñar una página web. Eso, no tiene una respuesta absoluta, todo depende.

Es muy común que en una página tengamos un sinnúmero de elementos que tengan alguna propiedad común (color, fondo, bordes, etc) y usando variables, nos permite definir esos valores una sola vez, darles un nombre personal y luego, usar `var()` con ese nombre de tal forma que si queremos hacer un cambio, basta modificar sólo un dato lo que ahorra tiempo y evita errores.

Cajas flexibles

El CSS3 incluye, en su fase experimental, el concepto de “cajas flexibles”, propiedades que permiten definir el diseño de tal forma de mantener la estructura en diferentes dispositivos y resoluciones de pantalla.

De alguna manera, tiende a reemplazar el uso de la propiedad **float** y simplifica la distribución de contenidos de tamaños desconocidos o variables.

La propiedad resumida se llama **flex** y tiene tres valores:

flex: grow shrink basis;

La propiedad **flex-grow** es un número positivo que determina la proporción en la que un elemento se re-dimensionará. Si todos los elementos tienen un valor de 1, el espacio se reparte igual entre ellos; si uno tiene un valor 2, este ocupará el doble de espacio que los otros.

La propiedad **flex-shrink** es opcional y debe ser un número o porcentaje positivo que define hasta qué valor puede reducirse el ancho en caso de ser necesario.

La propiedad **flex-basis** es opcional y debe ser una longitud o porcentaje que define el ancho por defecto.

Normalmente se utiliza la propiedad resumida y allí, definimos el tamaño relativo de cada elemento. Un par de ejemplos:

flex: 1 6 20%;

flex: 3 1 60%;

La propiedad resumida **flex-flow** define la dirección y la forma en que se distribuirán los contenidos flexibles:

flex-flow: direction wrap;

La propiedad **flex-direction** especifica la dirección en que serán mostrados los elementos flexibles y admite cuatro valores: **row** (de izquierda a derecha), **column** (de arriba hacia abajo) y los inversos **row-reverse** (de derecha a izquierda) y **column-reverse** (de abajo hacia arriba).

La propiedad **flex-wrap** es opcional y determina si el elemento se mostrará en una sola línea o no. Admite tres valores **nowrap** (es el valor por defecto) , **wrap** (multi-línea) y **wrap-reverse**.

En general, usamos la propiedad resumida de este modo:

flex-flow: row nowrap;

Si no indicamos nada, los contenidos flexibles se ubicarán tal como han sido enumerados (es lo que pasa normalmente con todas las etiquetas HTML) pero, podemos usar la propiedad **order** para indicar ese orden con un número:

order: 5;

Aunque existen otras, ya tenemos todas las propiedades necesarias para generar un ejemplo. Trataremos de diagramar una página tradicional donde hay un encabezado, un pie de página, un contenido principal y dos secciones laterales a su lado:

Este sería el HTML básico de un ejemplo:

```
<header></header>
<div id='contenedor'>
  <section id="izquierda"></section>
  <section id="derecha"></section>
  <article></article>
</div>
<footer></footer>
```

En este caso, tanto el encabezado como el pie de página serán bloques de altura fija pero de colores diferentes para reconocerlos fácilmente:

```
header, footer {display: block; height: 100px;}
header {background-color: aqua;}
footer {background-color: yellow;}
```

El elemento flexible será el llamado *contenedor* con tres columnas que también diferenciaremos mediante colores de fondo.

Le indicamos al navegador que ese elemento será una caja flexible utilizando la propiedad **display** con el valor **flex** y usaremos **flex-flow** para definir la forma en que se distribuirán aunque, como usaremos los valores por defecto no sería necesario agregarla:

```
#contenedor {
  display: flex;
  flex-flow: row nowrap;
  min-height: 400px; /* como carece de contenido debemos darle alguna altura */
}
```

Y ahora las tres columnas flexibles donde la central será el doble de ancho que las otras y usaremos **order** para indicar el orden de izquierda a derecha sin importar como hayamos escrito el código HTML:

```
#contenedor article {
  background-color: azure;
  flex: 3 1 50%; /* por defecto ocupará el 50% del ancho de la ventana */
  order: 2; /* será la segunda */
}
```

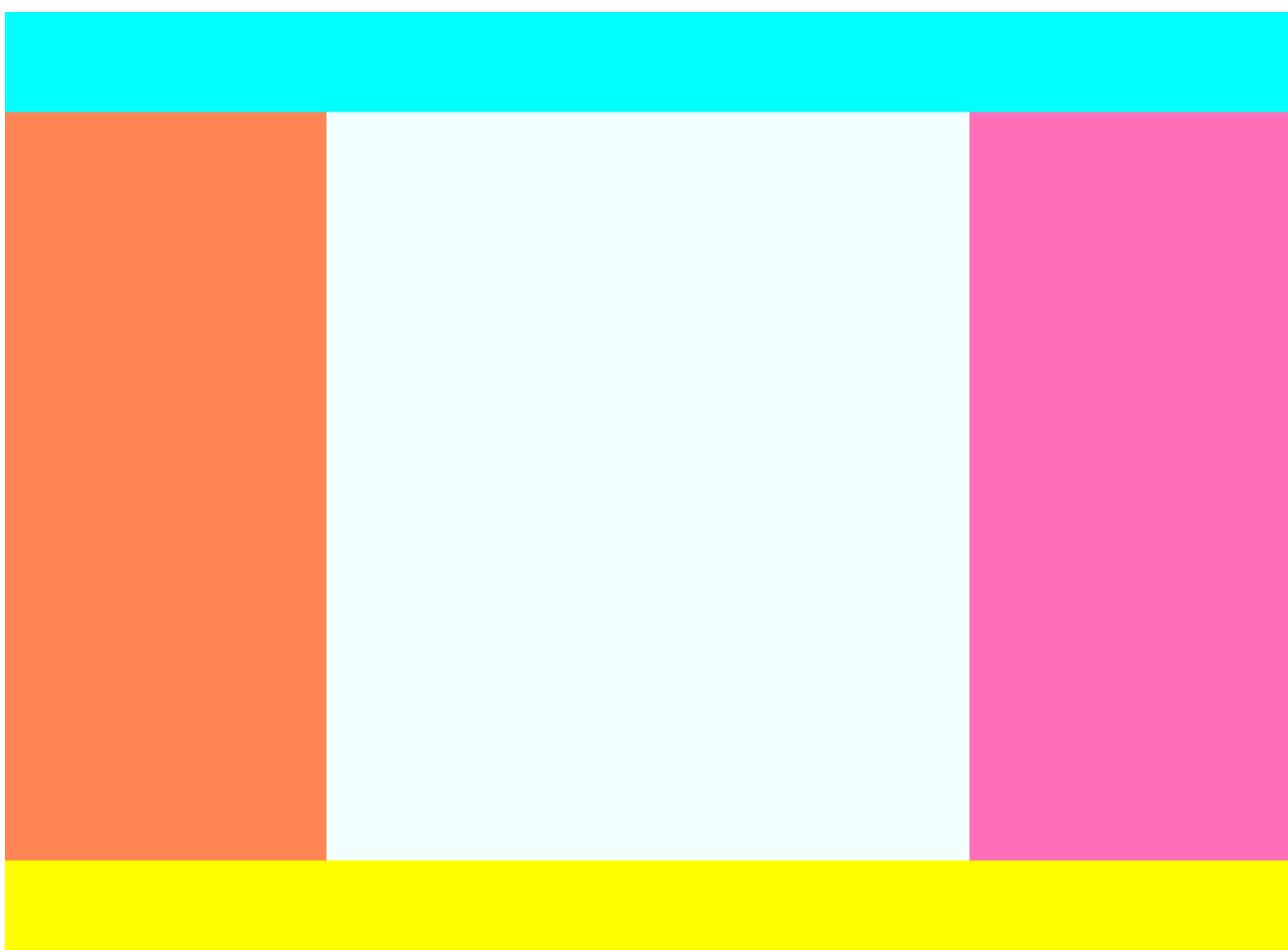
```

section#izquierda {
    background-color: coral;
    flex: 1 6 25%; /* por defecto ocupará el 25% del ancho de la ventana */
    order: 1; /* será la primera */
}

section#derecha {
    background-color: hotpink;
    flex: 1 6 25%; /* por defecto ocupará el 25% del ancho de la ventana */
    order: 3; /* será la tercera */
}

```

El resultado sería algo como esto:



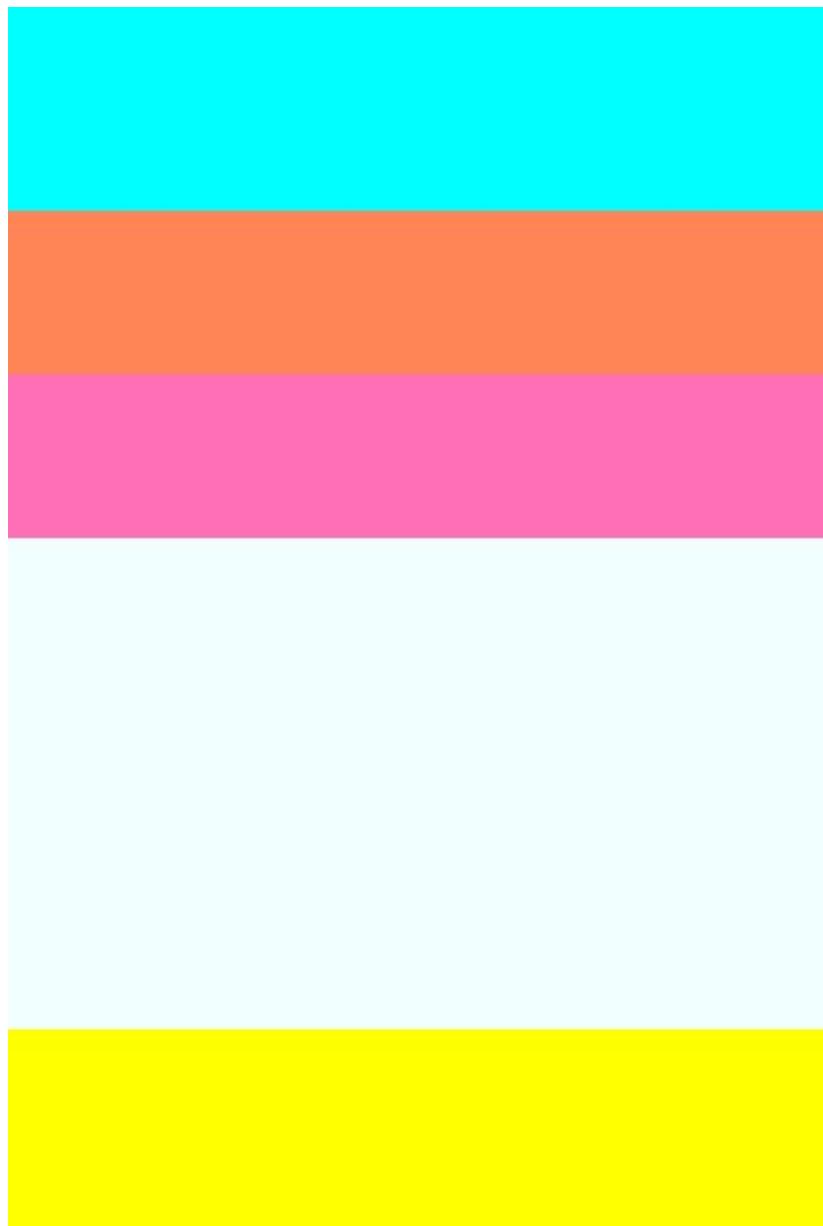
Ahora bien, supongamos que queremos que esa estructura se vea correctamente en un teléfono móvil o un dispositivo que tenga una pantalla angosta.

La respuesta rápida sería decir “eso es imposible” y es bastante cierta, para que esa página se vea correctamente, la estructura debería cambiar, deberíamos mostrarla en una sola columna, ocultar algunos contenidos secundarios, etc.

Todo eso lo podemos hacer utilizando `@media`, estableciendo alguna clase de límite y agregando reglas específicas.

```
@media all and (max-width: 480px) {  
  #contenedor{  
    /* modificamos la caja flexible para que se distribuya en una columna*/  
    flex-flow: column;  
  }  
  #contenedor article, section#izquierda, section#derecha {  
    /* eliminamos el orden */  
    order: 0;  
  }  
}
```

Y el modelo podemos probarlo en el mismo navegador (utilizando prefijos si es necesario) y re-dimensionando la ventana para ver los cambios.



Hay algunas otras propiedades asociadas con las cajas flexibles que nos permiten establecer el alineamiento de los contenidos.

La propiedad **align-content** define ese alineamiento cuando no “se llena” el espacio disponible y acepta los siguientes valores:

stretch se expanden hasta cubrirlo (es el valor por defecto)

flex-start se alinean al inicio del contenedor

flex-end se alinean al final del contenedor

center se alinean en el centro del contenedor

space-between los primeros se alinean al inicio del contenedor y los últimos al final

space-around se alinean dejando el mismo espacio entre ellos

Las propiedades **justify-content** y **align-items** definen el alineamiento horizontal y vertical de cada item y acepta los siguientes valores: **flex-start**, **flex-end** y **center**. A su vez, **justify-content** acepta **space-between** y **space-around** y **align-items** los valores **baseline** y **stretch**.

La propiedad **align-self** permite modificar la alineación definida por align-items para un item específico y los valores posibles son **auto**, **flex-start**, **flex-end**, **center**, **baseline** y **stretch**.

Entonces:

1. el CSS es como el ajedrez: lo elemental se aprende en un día y se puede pasar el resto de la vida tratando de dominarlo.
2. el CSS es como algunas religiones: la gente se vuelve fundamentalista con mucha rapidez.
3. el CSS es como una canción de los Beatles: simple y bella.



Referencias

Dudas y errores cuando se usa CSS

Cuando escribimos CSS los errores suelen ser sutiles pero existen. Sutiles porque el navegador los ignora, si algo está mal, simplemente sigue de largo. Eso, los hace más difíciles de detectar.

1. ¿Cuál es la diferencia entre estas dos reglas?

```
.demo {color: red;}  
p.demo {color: red;}
```

La primera, hará que el texto de cualquier etiqueta que tenga como atributo la clase *demo*, se vea de color rojo:

```
<div class="demo"> el texto es rojo </div>  
<span class="demo"> el texto es rojo </span>
```

La segunda, sólo funcionará en las etiquetas *p* que tengan ese atributo, cualquier otra, no será afectada.

```
<div class="demo"> el texto será de cualquier color </div>  
<p class="demo"> el texto es rojo </p>
```

2. ¿Y cuál es la diferencia entre estas dos reglas?

```
.demo p {color: red;}  
p.demo {color: green;}
```

La primera, hará que los textos de las etiquetas *p* que estén dentro de algún contenedor que tenga como atributo la clase *demo*, se vea de color rojo:

```
<div class="demo">  
  <p> el texto es rojo </p>  
  <span> el texto será de cualquier color </span>  
</div>
```

La segunda, hará que se vean rojos los textos de cualquier etiqueta *p* que tenga definida esa clase, tal como en el caso anterior.

3. ¿Cuál es la diferencia entre un ID y una clase?

¿Da lo mismo usar uno que otro? Definitivamente no. Los **ids** deben ser únicos, sólo una etiqueta en toda la página debe tener ese nombre; si se usan **ids** y clases en la misma etiqueta, las reglas definidas para el **id** tienen preponderancia sobre las de la clase porque las propiedades poseen un orden de prioridades:

```
#demoid {color: red;}  
.demoCLASE {color: green;}  
  
<div id="demoid" class="demoCLASE"> el texto será rojo </div>
```

4. ¿Puede usarse hover para modificar una etiqueta interna?

```
<div class="demo">
  <p> Sed vitae tortor turpis. <span>Nullam blandit</span> ornare urna vitae. </p>
  <p> Nam lacinia lacinia risus, sed <span>elementum libero</span> imperdiet. </p>
</div>
```

Con estas reglas, cuando ponemos el cursor sobre ese rectángulo, el texto será amarillo:

```
.demo {color: red;}
.demo:hover {color: yellow;}
```

Con esta otra, lo que cambiará es el color de las etiquetas `` que estén dentro del `<div>`:

```
.demo {color: red;}
.demo:hover span {color: yellow;}
```

5. ¿Los espacios son obligatorios?

Los espacios son obligatorios cuando separan palabras claves. Estas dos cosas son lo mismo:

```
.demo{color:red;}
.demo {color : red;}
```

Estas dos no son lo mismo:

```
.demo{background:trasnparent url() no-repeat left top;}
.demo{background:trasnparent url()no-repeat left top;}
```

La primera, funcionará correctamente en cualquier navegador, la segunda no porque falta el espacio entre el cierre del paréntesis y la palabra `no-repeat`.

6. ¿Qué propiedad se aplica cuando se definen varias veces?

```
.demo {
  color: red;
  font-size: 18px;
  color: green;
}
```

La última; siempre la última; en este caso, el color a usar será el verde.

7. ¿Hay que terminar siempre con punto y coma?

Deberíamos acostumbrarnos a que todas las propiedades terminen con un punto y coma aunque es cierto que en la última de una declaración no es obligatorio y que la que está antes de la llave de cierre no lo requiere pero, mejor colocarlo siempre hasta hacerse el hábito.

Esto es correcto:

```
.demo {  
    color: red;  
    font-size: 12px  
}
```

Esto es incorrecto:

```
.demo {  
    color: red  
    font-size: 12px  
}
```

8. ¿Qué pasa si el valor que utilzo no es aceptado?

Son muchos los casos donde se usan valores que no tienen efectos. El navegador los ignora porque ciertas propiedades no los admiten. Por ejemplo, **padding** no admite valores negativos ni la palabra **auto**.

Todos los valores de color en formato hexadecimal deben estar precedidos por el símbolo **#** y no debe haber un espacio entre este y el valor. También es importante recordar que esos valores pueden tener 3 o 6 caracteres pero, cualquier otra cantidad, será ignorada y no tendrá ningún efecto.

9. ¿Se pueden agregar etiquetas HTML dentro de hojas de estilo?

No debemos agregar etiquetas HTML dentro de las definiciones CSS por lo tanto, si se quieren colocar comentarios, no se debe usar la sintaxis HTML. Esto es un error:

```
<style>  
    body {color: red;}  
    <!-- este es un comentario -->  
    div {float: left;}  
</style>
```

Los comentarios simples dentro de las etiquetas **<style>** se escriben así:

```
<style>  
    body {color: red;}  
    /* este es un comentario */  
    div {float: left;}  
</style>
```

10. El CSS es correcto pero la página se ve mal

Como los navegadores utilizan **DOCTYPE** para definir el modo en que se mostrará la página, es posible que esa declaración sea errónea o anticuada.

Actualmente hay dos modos básicos, el llamado *Quirk mode* (no estándar) se utiliza para que haya la mayor compatibilidad posible con versiones viejas y de ese modo, las páginas diseñadas en el pasado se pueden ver con el estilo con que fueron creadas.

El llamado *Standard Mode* (modo estándar) es el que trata de seguir las reglas definidas por la W3C y, cualquier página nueva, debería diseñarse bajo ese estándar.

Ese modo estándar debería tener alguna declaración **DOCTYPE** de este tipo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"  
"http://www.w3.org/TR/html4/loose.dtd">
```

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

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

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

O, simplemente:

```
<!DOCTYPE html>
```

11. ¿Se puede eliminar el valor de una propiedad o restaurarla a su valor por defecto?

No, eso es imposible. Lo que debería hacerse para lograr eso es volver a declararla aunque, si nos encontramos en esa situación significa que no hemos escrito las reglas de modo lógico y es probable que hayamos definido estilos que son innecesarios.

12. ¿Es mejor usar propiedades abreviadas?

No es ni mejor ni peor, a veces, como en el caso de **border**, casi siempre las definimos abreviadas con todos sus parámetros; en otros como **font**, parecería más sencillo y más flexible declararlas una por una. Al usar una propiedad abreviada debemos tener en cuenta que todos los parámetros serán agregados a su valor por defecto, los hayamos enumerado o no. Por ejemplo:

```
background: white;
```

El navegador lo interpreta como:

```
background: white none repeat scroll 0 0;
```

13. ¿Se deben utilizar servicios que ofrecen optimizar el CSS?

Optimizar ¿Quién no quiere optimizar las cosas?

Son muchas los sitios que nos hablan de comprimir el CSS e incluso hay sitios que ofrecen hacerlo pero esas cosas automáticas hay que utilizarlas con cuidado; con mucho cuidado.

VER REFERENCIAS [Optimizar el CSS]

El ID siempre debe ser único

¿Quién es quién? Este es Juan y aquel es Pedro; más fácil, este es Juan y aquella es Antonia. Facilísimo. No es necesario saber sus nombres para identificarlos y no confundirlos pero, pese a lo que muchos creen, las computadoras son idiotas. Es decir, hacen cuentas muy pero muy rápido pero ignoran todo lo que no les explicamos con pelos y señales.

Ese es el problema con el atributo **id** que tanto se utiliza y que tantos problemas causan cuando ponemos dos cosas con el mismo nombre y sólo funciona una de ellas.

El atributo **id** se utiliza para IDENTIFICAR algo y ese algo debe ser ÚNICO, sólo debe haber uno con ese nombre en toda la página; si no lo es, el navegador no sabrá a cuál nos referimos cuando le decimos que haga algo. Por eso, los IDs no se ponen en todas las etiquetas, sólo los usaremos cuando sea necesario y la regla genérica dice que deberíamos reservarlos exclusivamente para los sectores importantes, para definir los grandes rectángulos que conforman un sitio web: el *header*, las columnas, el *footer*, etc.

Pero, también los usamos cuando requerimos identificar sectores auxiliares, cualquier cosa que se nos ocurra manipular con JavaScript.

Vamos a un ejemplo que no funcionará. Una función simple que permuta la visibilidad de una etiqueta. Dado su **id**, si está oculta se muestra y si está visible se oculta:

```
<script>
    function toggletag(cual) {
        var elElemento = document.getElementById(cual);
        if(elElemento.style.display == 'block') {
            elElemento.style.display = 'none';
        } else {
            elElemento.style.display = 'block';
        }
    }
</script>
```

Ponemos dos etiquetas **<div>** y queremos ocultar el segundo pero ... en ambos hemos usado el mismo **id**:

```
<div id="mi_id" style="display:block;"> Este es un DIV cualquiera ... </div>
<div id="mi_id" style="display:block;"> Este es el DIV que quisiera ocultar ... </div>
<a href="javascript:toggletag('mi_id');">click</a>
```

No funciona. Se oculta el primero y no el segundo ya que el navegador lo encontró primero e ignora al otro. Todo se debe a ese detalle, ambos tienen el mismo **id** así que, para que esto funcione, deberíamos ponerles nombres diferentes:

```
<div id="mi_id_1" style="display:block;"> Este es un DIV cualquiera ... </div>
<div id="mi_id_2" style="display:block;"> Este es el DIV que quisiera ocultar ... </div>
<a href="javascript:toggletag('mi_id_2');">click</a>
```

Sobre IDs y clases

Algo que siempre causa alguna confusión cuando escribimos CSS es determinar si debemos aplicar esas definiciones como **class** o **id** ¿Cuál es la diferencia?

Tanto una como otra sirven para identificar una etiqueta y, de ese modo, nos resulta más sencillo agregarle propiedades. Por ejemplo:

```
<style>
    #nombreID {color: red;}
    .nombreClase {color: blue;}
</style>

<div id="nombreID"> este texto será rojo </div>
<div class="nombreClase"> este texto será azul </div>
```

Para definir las propiedades de un **id** le anteponemos el símbolo **#** al nombre y para definir una clase, usamos un punto.

Una regla elemental y que no debe violarse dice que sólo usaremos un **id** si esa etiqueta es única, no debería haber dos etiquetas con el mismo **id** en la misma página; en cambio, usaremos **class** si es un estilo que repetiremos en diferentes etiquetas.

Además, suele decirse que el atributo **id** se utiliza para definir áreas específicas de una página (el *header*, el *footer*, un menú, etc) y una clase se usa para definir estilos de tipo general (enlaces, listas, etc).

Los **id** nos permiten organizar el estilo e identificar etiquetas para luego manipularlas con JavaScript. Las clases, nos evitan escribir códigos repetidos y esa es su mayor utilidad.

Hay varias formas de implementarlas; esto hará que los enlaces que contengan esa clase, se muestren de color verde:

```
a.green {color: green;}

<a class="green"> el enlace será de color verde </a>
<p class="green"> este párrafo NO será de color verde </p>
```

En cambio, esta otra hará que cualquier etiqueta que contengan esa clase, se muestren de color verde:

```
.green {color: green;}

<a class="green"> el enlace será de color verde </a>
<p class="green"> este párrafo también será de color verde </p>
```

Tanto las clases como los **id** pueden combinarse así que no es extraño ver cosas como estas:

```
<div id="unNombre" class="unaClase"> ..... </div>
```

O bien:

```
<div class="unaClase otraClase"> ..... </div>
```

¿Para qué usamos **id** y **class** juntas? Por ejemplo:

```
<style>
  .ejemploCSS {
    background-color: #000;
    border: 1px solid #CCC;
    float: left;
    height: 50px;
    margin-right: 10px;
    padding: 10px;
    width: 150px;
  }
  #ejemploCSS1 {color: white;}
  #ejemploCSS2 {color: yellow;}
  #ejemploCSS3 {color: red;}
</style>
```

Tenemos una clase donde hemos definido una serie de propiedades como fondo, bordes y tamaño que aplicaremos a un **<div>** así que, si colocamos tres, uno al lado del otro, se verán iguales.

Ahora bien, le agregaremos un **id** a cada uno de ellos para darles una propiedad exclusiva, el color del texto:

```
<div id="ejemploCSS1" class="ejemploCSS"> ... </div>
<div id="ejemploCSS2" class="ejemploCSS"> ... </div>
<div id="ejemploCSS3" class="ejemploCSS"> ... </div>
```

¿Y qué pasa si hay propiedades contradictorias? Por ejemplo, definimos el color de fondo tanto en la clase como en el **id** ¿Cuál se verá? ¿será negro como dice la clase o variará como dicen los **id**?

```
<style>
  .ejemploCSS {background-color: #000;}
  #ejemploCSS1 {background-color: white;}
  #ejemploCSS2 {background-color: yellow;}
  #ejemploCSS3 {background-color: red;}
</style>
```

El fondo será el del **id** y el de la clase será ignorado.

Claro, uno puede decir, lo que ocurre es que, primero le decimos que es negro y luego le decimos que es de otro color, lo sobrescribimos, es lógico pero, veamos si lo escribimos al revés.

```
<style>
  #ejemploCSS1 {background-color: white;}
  #ejemploCSS2 {background-color: yellow;}
  #ejemploCSS3 {background-color: red;}
  .ejemploCSS {background-color: #000;}
</style>
```

Ahora ponemos el fondo negro al final ¿qué se verá? Pues, lo mismo ya que, para decirlo de alguna manera, un **id** es “más importante” que una clase. Para hacer que una clase “sobrescriba” una propiedad de un **id**, le agregamos la palabra **!important**:

```
<style>
  .ejemploCSS {background-color: #000 !important;}
  #ejemploCSS1 {background-color: white;}
  #ejemploCSS2 {background-color: yellow;}
  #ejemploCSS3 {background-color: red;}
</style>
```

Las propiedades CSS por defecto

Toda propiedad CSS tiene un valor, lo hayamos indicado de modo explícito o no.

Si el valor ha sido definido en una regla de estilo, será ese el que usará el navegador; si no ha sido definido, usará el valor establecido por su contenedor (el elemento padre) y, si este último tampoco ha sido definido, usará un valor por defecto.

El `<body>` de una página es donde están definidas por defecto algunas de esas propiedades.

Es más o menos conocido que el fondo (**background**) de una página es de color blanco y que el color de los textos es negro pero tiene algunas más que suelen perturbar, por ejemplo, el margen (**margin**):

```
body {  
    background: #FFF url() repeat left top;  
    color: #000;  
    font-family: Times New Roman;  
    font-size: 16px;  
    margin: 8px;  
}
```

Los enlaces también tienen propiedades por defecto, son de color azul y se muestran subrayados.

Todas las etiquetas de títulos tienen fuentes en negrita (**font-weight:bold**) y además, un tamaño que está en relación al tamaño de la fuente definida en el `<body>`. Para colmo, también tiene márgenes:

```
h1 font-size: 2em; margin: 21px 0;  
h2 font-size: 1.5em; margin: 19px 0;  
h3 font-size: 1.2em; margin: 19px 0;  
h4 font-size: 1em; margin: 21px 0;  
h5 font-size: .8em; margin: 21px 0;  
h6 font-size: .7em; margin: 26px 0;
```

Ni las etiquetas `<div>` ni `` tienen propiedades por defecto pero si las tiene la etiqueta `<p>`; esta posee un margen superior e inferior igual al tamaño de la fuente por defecto:

```
p {margin:1em 0;}
```

Más etiquetas con propiedades pre-definidas:

```
blockquote margin: 16px 40px;  
pre font-family: monospace; margin: 16px 0;
```

Las imágenes que sirven como enlaces tienen un borde de color igual al color de los enlaces:

```
img {border: 2px solid #0000EE;}
```

Probablemente, las etiquetas que causan más conflictos son las listas ya que tanto `` como `` tienen varias propiedades incluyendo márgenes y separaciones:

```
ol margin:16px 0; padding-left: 40px; list-style-position: outside; list-style-type: decimal;  
ul margin:16px 0; padding-left: 40px; list-style-position: outside; list-style-type: disc;
```

Es muy común leer que para evitar que todas estas propiedades no nos molesten, lo mejor es “resetearlas”, es decir, poner al inicio de nuestra plantilla, algunas definiciones que “limpien” esas cosas. Hay decenas de ejemplos que pueden verse *online* pero, para un uso normal, no es necesario tanto, basta agregar unas pocas, justo al inicio de la hoja de estilo:

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

```
html, body {height: 100%;}  
body {height: 100%;}
```

```
a, a:visited, a:link, a:active {outline: none; text-decoration: none;}  
a:hover {outline: none; text-decoration: none;}  
a img {border: none; outline: none; text-decoration: none;}
```

```
object, embed {outline: none;}
```

```
ol, ul, li {list-style: none;}
```

Por supuesto, esto tendrá que verse caso por caso ya que es posible que algunas de las propiedades por defecto nos sean útiles y por lo tanto, no nos interese sacarlas pero, siempre es bueno saber que allí están y, cuando comenzamos un sitio nuevo, lo mejor es eliminarlas a todas y tomarnos el trabajo de definirlas una por una.

Un espacio hace la diferencia

Todos sabemos que los caracteres espacio son importantes; separan las palabras y evitar que eso que *escribimos sea una局限a silegible*.

En CSS pasa lo mismo, no tanto porque no pueda leerse eso que escribimos sino porque al no estar donde deben estar, el resultado es un error. El caso típico es el de la propiedad **background**; si escribimos esto:

```
background:#FFF url(uná_imagen)no-repeat left top;
```

En algunos dispositivos no se verá la imagen y no es un error del navegador, es que lo hemos escrito mal, sin dejar el espacio entre el cierre del paréntesis y la palabra **no-repeat**

Pero no sólo se producen errores en tal o cual navegador, es algo que podría ocurrir en todos si, por ejemplo, queremos poner una regla de estilo para que una etiqueta **** tenga un determinado formato (color rojo en este caso):

```
<div id="demo" class="ejemplo">  
    un texto cualquiera <span>esto en rojo</span> un texto cualquiera  
</div>
```

¿Cuál debería ser la regla de estilo?

```
#demo.ejemplo span {color:#F00;} /* ¿esta? */  
#demo .ejemplo span {color:#F00;} /* ¿o esta otra? */
```

Si usamos la primera, todo estará bien; si usamos la segunda, no pasará nada. Esto, se debe a que el espacio indica “algo”.

En la primer, traducida al español, se indica que en el elemento cuyo **id** es **#demo** y cuya clase es **.ejemplo**, las etiquetas **** interiores son de color rojo. Por eso, no hay un espacio intermedio.

En la segunda decimos que en el elemento cuyo **id** es **#demo**, en toda etiqueta interna que sea de la clase **.ejemplo**, las etiquetas **** internas serán de color rojo. Para que esto último funcionara, el HTML debería tener una estructura diferente:

```
<div id="demo">  
    <div class="ejemplo">  
        un texto cualquiera <span>esto en rojo</span> un texto cualquiera  
    </div>  
    <div class="otra">  
        un texto cualquiera <span>esto no será rojo</span> un texto cualquiera  
    </div>  
</div>
```

Por supuesto, la primera parece redundante y no siempre es necesario escribirlas de ese modo indicando el `id` y la clase; estas dos reglas también harían lo mismo en el primer ejemplo:

```
#demo span {color:#F00;}  
.ejemplo span {color:#F00;}
```

Y esta haría lo mismo en el segundo ejemplo:

```
.ejemplo span {color:#F00;}
```

No todo funcionará igual, las reglas también dependen de lo que se llama prioridad por lo que ciertas definiciones no son tenidas en cuenta ya que hay otras que indican lo contrario y para el navegador, son más “importantes”.

Un ejemplo:

```
#demo.ejemplo span {color:#F00;} /* color rojo */  
#demo span {color:#FF0;} /* color amarillo */  
.ejemplo span {color:#0F0;} /* color verde */
```

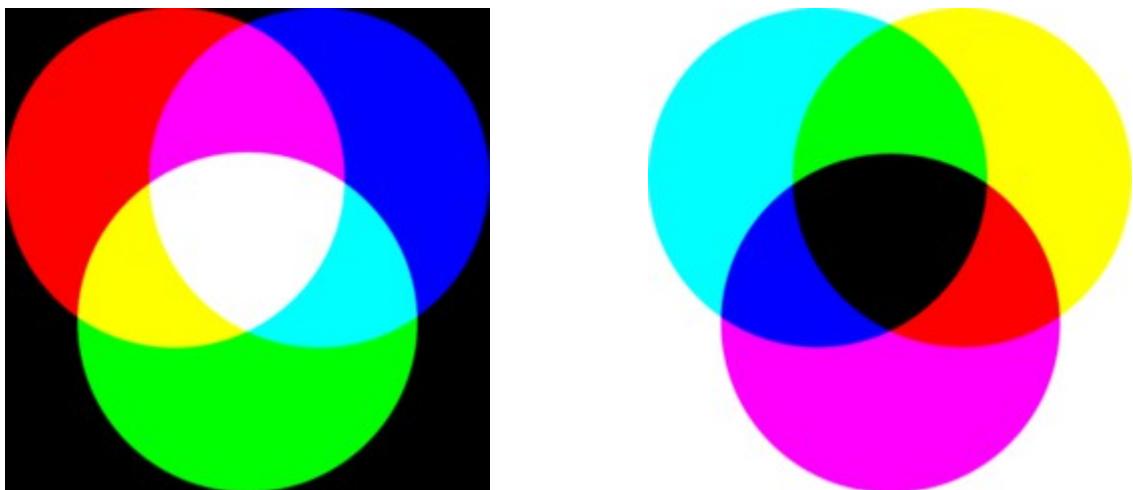
La etiqueta `` se vería de color rojo pese a que las otras dos reglas dicen lo contrario; si elimino la primera se vería amarillo así que si quisiera que se viera verde, sólo debería incluir la última o agregar la palabra `!important` a alguna de ellas para forzar al navegador a prestar atención.

Algo sobre colores

En la web, un color es un valor expresado como RGB o una palabra que lo identifica y que tienen su origen en los nombres dados a los 16 colores de la paleta VGA original de allá lejos y hace tiempo: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white y yellow.

Los colores pueden definirse con tres números uno para el rojo (R=red), uno para el verde (G=green) y otro para el azul (B=blue) que representan la intensidad de sus componentes.

Este sistema llamado RGB utiliza números que pueden variar entre 0 y 255 (un byte) o sea que, cada componente puede tener 256 valores distintos lo que, al combinarse ($256 \times 256 \times 256$) nos da una gama de 16 millones de colores (el llamado color verdadero).



Los colores primarios son estos:

`rgb(255,0,0) = #FF0000 = Rojo`
`rgb(0,255,0) = #00FF00 = Verde`
`rgb(0,0,255) = #0000FF = Azul`

Los colores secundarios son:

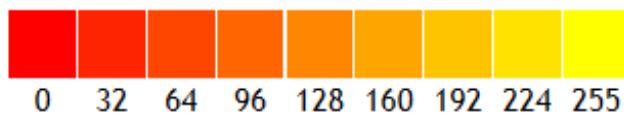
`rgb(0,255,255) = #00FFFF = Cyan`
`rgb(255,0,255) = #FF00FF = Magenta`
`rgb(255,255,0) = #FFFF00 = Amarillo`

Los tonos de gris van desde el blanco (todos los colores con intensidad máxima) y el negro (todos los colores con intensidad mínima), es decir van desde `rgb(0,0,0)` hasta `rgb(255,255,255)`.

Para crear tonos de gris, la técnica a utilizar es sencilla, los tres valores de RGB deben ser iguales:

$\text{rgb}(0,0,0) = \#000000$ = Negro
 $\text{rgb}(64,64,64) = \#404040$ = Gris Oscuro
 $\text{rgb}(128,128,128) = \#808080$ = Gris
 $\text{rgb}(192,192,192) = \#\text{COCOCO}$ = Gris Claro
 $\text{rgb}(255,255,255) = \#\text{FFFFFF}$ = Blanco

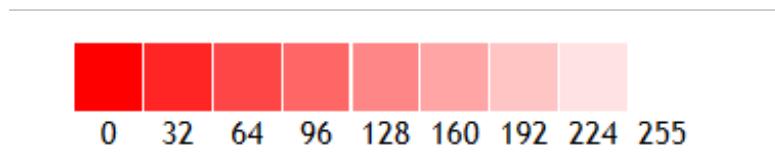
Para conseguir “tonos” de un determinado color, usamos simple aritmética; si partimos del rojo $\text{rgb}(255,0,0)$ y colocamos valores progresivos en el componente verde, obtendremos una gama:



Lo mismo pasaría si variáramos el otro componente, el azul:



O ambos componentes simultáneamente:



Para los colores puros (saturados) se combinan sólo dos colores primarios: $\text{rgb}(X,Y,0)$, $\text{rgb}(Y,X,0)$, $\text{rgb}(0,X,Y)$ o $\text{rgb}(0,Y,X)$ donde X no es cero y es mayor o igual que Y. Entonces:

- si $Y = 0$ es un color primario.
- si $X = Y$ es un color secundario
- si $Y = X / 2$ es un intermedio entre un color primario y un color secundario
($\text{rgb}(255,128,0) = \text{naranja}$)
- si $Y < X / 2$ es un intermedio pero más cercano al color primario
($\text{rgb}(255,64,0) = \text{naranja rojizo}$)
- si $Y > X / 2$ es un intermedio pero más cercano al color secundario
($\text{rgb}(255,192,0) = \text{naranja amarillento}$)

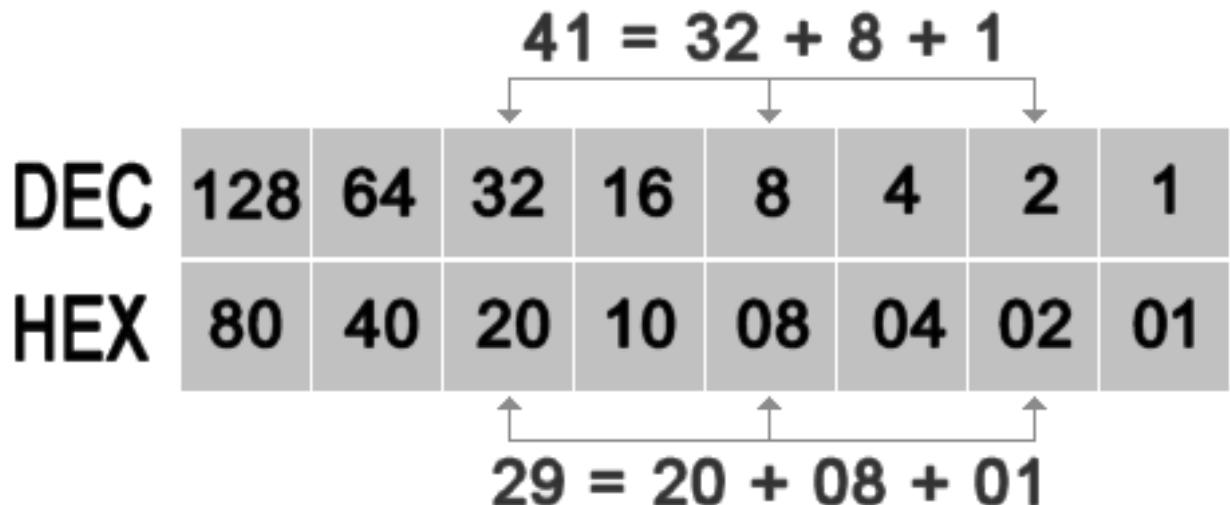
Los colores expresados como RGB pueden ser escritos de cuatro formas diferentes:

- como un valor hexadecimal doble: $\#00CC00$ que varía entre 000000 y FFFFFF
- como un valor hexadecimal simple: $\#0C0$ que varía entre 00 y FF
- como un valor decimal: $\text{rgb}(0,204,0)$ donde cada valor puede variar entre 0 y 255
- como un porcentaje: $\text{rgb}(0\%,80\%,0\%)$ donde cada valor puede variar entre 0.0 y 100.0

Aunque cada uno elegirá el tipo de escritura que le resulte más cómodo, el formato hexadecimal es el más empleado. Los números hexadecimales, en lugar de utilizar 10 símbolos como el sistema decimal (0 ... 9) utilizan 16 (0 1 2 3 4 5 6 7 8 9 A B C D E F); así, en lugar del número 10, usamos el número A o 0A.

En un color hexadecimal, cada componente está representado por dos dígitos, por ejemplo:

#F08000 = F0 80 00



En la web, hay una gama de colores llamados "seguros" es decir, que no serán distorsionados por las diferentes tarjetas de vídeo y se compone de 216 colores que resultan de todas las combinaciones posibles de seis valores 0, 51, 102, 153, 204 y 255 (00, 33, 66, 99, CC y FF) para cada uno de los componentes rojo, verde y azul ($6^3 = 216$) ¿Por qué esos números? Porque representan porcentajes de intensidad exactos (0%, 20%, 40%, 60%, 80% y 100%)

Como los monitores podrían manejar paletas de hasta 256 colores, esos 40 colores sin usar, Windows los reserva para identificar 20 colores de sistema del sistema operativo y deja otros 20 para que pueda definir el usuario en cada aplicación.

Esos "colores seguros", tienen "nombres así que en lugar de usar valores RGB podemos usar esos nombres. Así que es cualquiera de estos tres códigos, indica un color blanco:

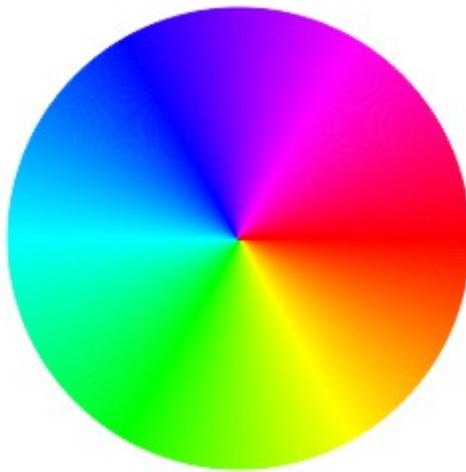
```
color: #FFFFFF;
color: rgb(255,255,255);
color: white;
```

Es probable que cualquiera que haya utilizado algún software de edición de imágenes se haya topado con este tipo de definiciones pero, aunque no sea un formato muy utilizado, veamos lo elemental.

Seguramente han visto una rueda de colores. Allí, cada grado es un color y el matiz (hue) es un número de 0 a 359 (los 360 grados) de la rueda.

En esa rueda, el color rojo está en 0° , el verde en 120° y el azul (240°).

Los otros dos valores son la saturación (saturation) y la luminosidad (lightness).



Conociendo el valor *hue* de un color podemos calcular el color complementario:

rojo = hsl(0,100%, 50%)

Como *h* (hue) es cero, el opuesto será 180:

cyan hsl(180,100%,50%)

La saturación de un color se representa como porcentaje por lo que puede variar entre 100% a 0% (donde 0 equivale a grises).

La luminosidad también se representa como porcentaje. Allí, el 100% es blanco y el 0% negro.

Un poco más allá de las imágenes

Una “rareza” del CSS es que tiene la posibilidad de manejar las imágenes como capas (*layers*) lo que permite que se “deslicen” unas sobre otras, creando efectos.

La técnica no es una novedad, se viene usando para crear juegos desde tiempo inmemorial pero, su aplicación a las páginas web abre nuevas posibilidades.

Vayamos despacio, lo primero que debería quedar claro son las propiedades CSS que permiten controlar el fondo de un elemento:

background: color image repeat attachment position

Esa es la forma rápida de establecer todas las propiedades simultáneamente. De todas ellas, la que nos va a interesar es la posición.

La propiedad **background-position** define la posición del fondo de un elemento. En CSS, la propiedad se establece con:

background-position: laPosicion

Y en JavaScript:

objeto.style.backgroundPosition = valor

donde *valor* es una cadena de texto que contiene uno o dos valores que puede ser:

una longitud: un valor expresado en cm, mm, in, pt, pc, px, em o ex

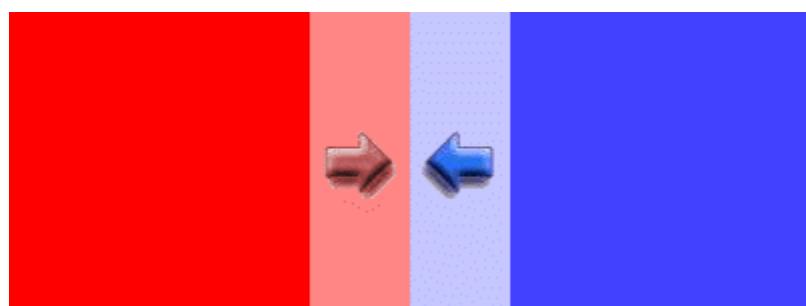
un porcentaje: valor del ancho o la altura del elemento)

el alineamiento vertical: top, center, bottom

el alineamiento horizontal: left, center, right

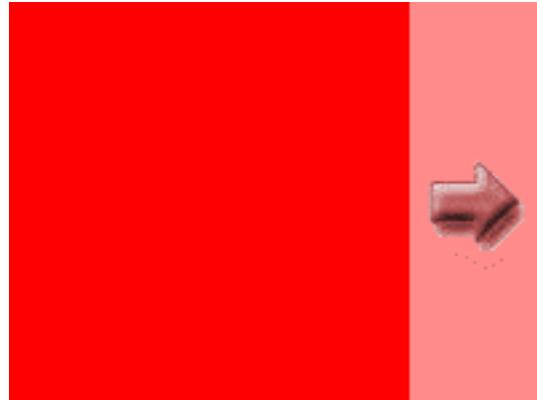
El valor por defecto es 0% 0% (left top). El primero especifica la coordenada horizontal y el segundo, la vertical pero, si sólo se establece solo uno de ellos, el mismo se aplica a ambos alineamientos.

Imaginemos esto, dos imágenes cualquiera, una al lado de la otra y formando parte del mismo archivo:



Ahora, supongamos que tenemos un `<div>` al que le asignamos una propiedad `background` para que se vea un fondo y le damos un tamaño exactamente igual a la mitad del ancho de la imagen:

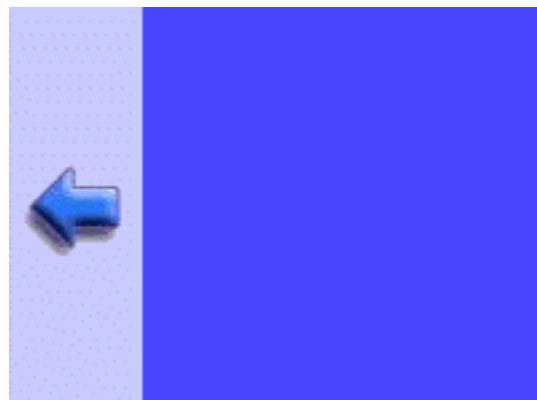
`background: transparent url(URL_imagen) no-repeat left top;`



Como el elemento contenedor `<div>` es más angosto que la imagen de fondo, sólo se verá la mitad izquierda.

Si cambiáramos un poco la propiedad, podría verse la otra mitad:

`background: transparent url(URL_imagen) no-repeat right top;`



Quiere decir que modificando la posición, con sólo un archivo, podemos mostrar dos imágenes.

¿Qué ventaja tiene esta técnica? que la imagen sólo se carga una vez, por lo tanto, la hace ideal para utilizarla como efecto *rollover* pero nada impide que haya más imágenes.

Si en lugar de utilizar las palabras claves de alineación, utilizamos longitudes, las posibilidades se tornan infinitas.

En este ejemplo tenemos una imagen modulada, los íconos de 32x32 se colocan, horizontalmente, uno a continuación del otro. De esta manera, una aritmética simple nos dice que, cada uno de ellos está a una distancia específica del borde izquierdo de la imagen: 0, 32, 64, 96, 128, 160, 192, 224, 256, 288 píxeles.



Así que, para hacer referencia a la imagen ubicada en el 7º lugar, el evento **onclick** será algo así:

```
onclick="getElementById('demo').style.backgroundPosition=-192px 0px";
```

donde el valor es negativo porque la imagen se desliza hacia la izquierda:



Una posibilidad más avanzada sería crear una animación, desplazando la imagen pixel por pixel.

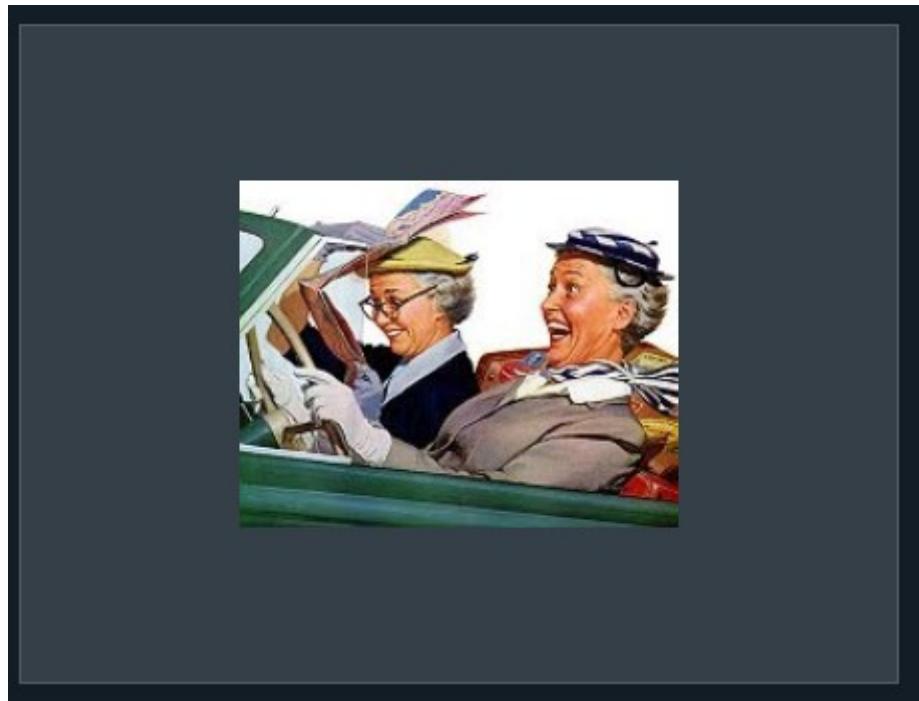
[VER REFERENCIAS \[Sobre el uso de los sprites\]](#)

Múltiples fondos con CSS3

Otra de las novedades del CSS3 es la posibilidad de utilizar varias imágenes de fondo en un mismo elemento. Hasta ahora, cuando colocamos un fondo a una etiqueta, el rectángulo puede contener sólo una y eventualmente, un color alternativo. Por ejemplo, un `<div>` así:

```
<div style=" background: #303941 url(URL_imagen) no-repeat 50% 50%; height: 300px; width: 300px;"></div>
```

Nos mostrará esto:



Es decir, un rectángulo de cierto tamaño donde la imagen de fondo esta centrada y como es pequeña, no alcanza a cubrir ese área así que el resto se llena con el color alternativo. Si quisiéramos poner una segunda imagen de fondo, deberíamos agregar otro rectángulo encima:

```
<div style=" background: #303941 url(URL_imagen1) no-repeat left top; height: 300px; width: 300px;">
  <div style="background: transparent url(URL_imagen2) no-repeat right bottom; height: 300px; width: 400px;"></div>
</div>
```

Una imagen se posiciona en el ángulo superior y otra en el ángulo inferior. Como en el `<div>` interno el color alterno se define como transparente, la segunda imagen se superpone a la primera.

Pero, la w3org propone simplificar esto y permitir que se coloquen varias imágenes en el mismo elemento. Para esto, es más cómodo separar la definición en partes; en lugar de:

```
background: color url(URL_imagen) repeat posicionX posicionY;
```

escribiremos cada una por separado, separando los datos con comas:

```
background-image: url(URL_imagen1), url(URL_imagen2);
```

```
background-position: left top, right bottom;
```

```
background-color: #303941;
```

```
background-repeat: no-repeat;
```

En la primera establecemos la dirección de dos imágenes distintas, en la segunda, la posición de cada una de ellas y en las dos últimas, como esas propiedades serán iguales para ambas, escribimos una sola.



Algunos críticos dicen que esto sólo aumentará el peso de las páginas pero, es una opinión que no tiene sustento ya que aún sin esta facilidad, nada impide agregar muchos fondos, esta posibilidad, sólo lo hace menos engorroso. También se dice que hay una limitación de ocho alternativas dadas por las combinaciones de *left top right y bottom* pero es un error, no está indicado si hay una limitación y la posición puede ser definida en pixeles así que es posible colocar un número indeterminado de fondos.

Los misterios de background-size

La alternativa de re-dimensionar una imagen de fondo está contemplada en el CSS3 mediante la propiedad **background-size** que nos permite establecer la dimensión de la imagen a usar como fondo y admite dos valores, el ancho y el alto, tanto en pixeles como en porcentajes.

Eventualmente, hay otros valores que podemos usar como **auto**, **contain** y **cover** que establecen la relación ancho/alto a mantener.

Supongamos una imagen de 200x158 pixeles que aplicaremos como fondo a un **<div>** al que le daremos una dimensión de 300x300. Agregado el estilo, podríamos tener este código:

```
<div style="background: #000 url(URL_imagen) no-repeat left top; height: 300px; width: 300px;"></div>
```

Y veríamos esto:



Como la imagen de fondo es pequeña, por defecto, no alcanza a cubrir el total del rectángulo así que, le agregaremos la nueva propiedad con valores de 100% tanto para el ancho como para el alto:

```
background-size: 100% 100%;
```

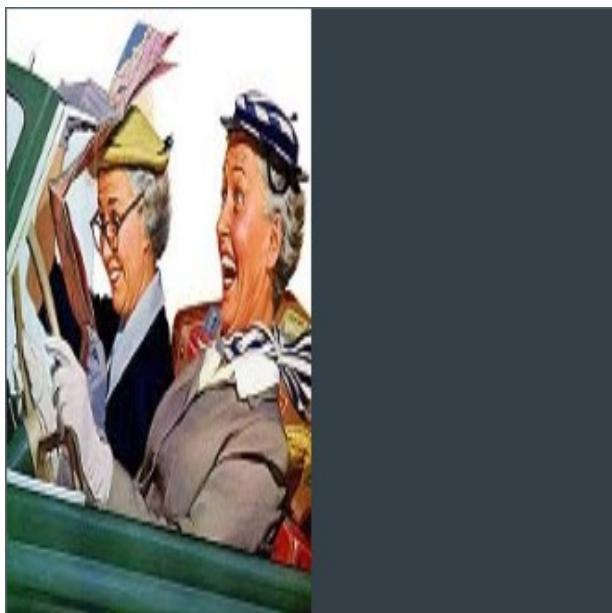
Y veríamos esto:



Ahora, hagamos lo mismo pero, con valores del 50% para el ancho en un caso y para el alto en el otro:

background-size: 100% 50%;
background-size: 50% 100%;

Este sería el resultado:



¿Y que hacen **contain** y **cover**? Los definimos así:

background-size: contain; /* el rectángulo de la izquierda */

background-size: cover; /* el rectángulo de la derecha */

Y veríamos esto:



Los misterios de background-clip

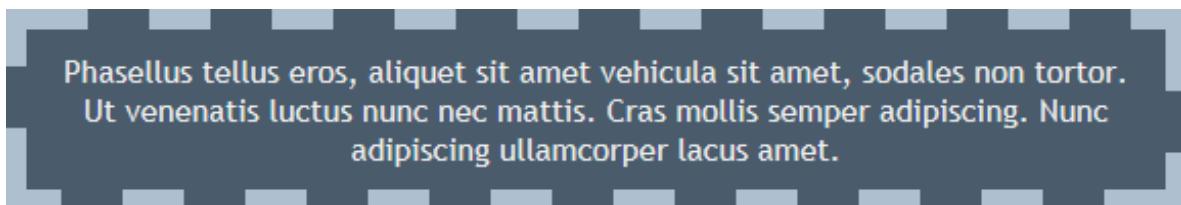
Los navegadores interpretan las propiedades CSS como se les da la gana, algunos de una forma, otros de otra; con el tiempo, esto ha mejorado bastante aunque siguen habiendo diferencias. Para colmo, a esas propiedades les agregan extensiones, funciones o características propias que no son parte de los estándares o que son experimentales.

Si bien es cierto que no se aplicarán a todos, nada impide que las usemos siempre y cuando chequeemos nuestro sitio y verifiquemos que se ve de manera más o menos razonable en cualquiera de ellos.

Una de ellas es la llamada **background-clip** que acepta dos valores: **border** y **padding**:

Esta propiedad, controla la forma en que se muestran los bordes y los fondos de cualquier elemento. Por defecto, el valor que tiene es **border** y lo mejor es verlo en acción. Colocamos entonces una etiqueta con un borde exagerado para que el detalle sea bien visible:

```
<div style="background-color:#456; border: 10px dashed #ABC;">  
    cualquier contenido  
</div>
```

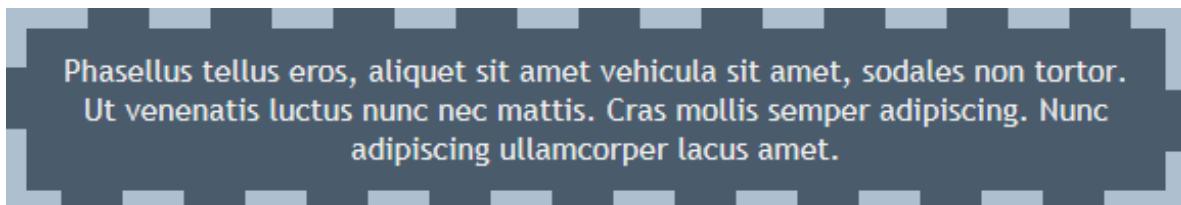


Lo mismo ocurrirá con una imagen o con cualquier otro contenido:



Así se ve por defecto, tenga o no tenga agregado **background-clip: border**, el fondo se extiende por debajo del borde cuando este tiene partes transparentes. Ahora, cambiemos ese valor por **padding**:

```
<div style="background-color:#456; border: 10px dashed #ABC; background-clip: padding;">  
    cualquier contenido  
</div>
```



El resultado es diferente, el fondo ya no se extiende debajo del borde.



Sobre el uso de los sprites

¿Será más rápido cargar una imagen grande o varias imágenes pequeñas?

No es una pregunta retórica, durante mucho tiempo, siempre se ha creído lo segundo y de hecho, la técnica del *slice*, partir una imagen en pedazos y cargar cada parte por separado, es algo que incluso poseen varios editores de imágenes como Photoshop o Fireworks.

Sin embargo, de un tiempo a esta parte, la idea ha ido cambiando y en estos momentos, hay una tendencia a lo contrario, al uso de los llamados *sprites*, es decir combinar varias imágenes en un solo archivo y luego, utilizar CSS para mostrarlas.

¿Cómo hacemos esto? Es simple, sólo necesitamos un poco de aritmética.

Por ejemplo, supongamos que tengo tres íconos de 16x16 que quiero mostrar como enlaces y cada uno de ellos, tiene un efecto *hover*; necesitaría seis imágenes individuales. Las combino con un editor, poniendo una al lado de la otra y creo una sola imagen que tendrá 48x32:



En la fila superior tengo los tres íconos en estado "normal" (los que veré por defecto) y abajo los tres que mostraré cuando pase el ratón sobre ellos.

La propiedad **background** nos permite colocar una imagen de fondo y posicionarla, es decir, que si dimensionamos algo, podemos agregarle un fondo de mayor tamaño y sólo mostrar una parte de esa imagen. Creamos unas clases CSS para que se vea el ejemplo:

```
/* las propiedades generales */
a.iconos {
    background: transparent url(URL_IMAGEN) no-repeat 0px 0px;
    display: block;
    height: 16px;
    width: 16px;
}

/* las propiedades de cada uno de los íconos */
a.icono1 {background-position: 0px 0px;}
a.icono2 {background-position: -16px 0px;} /* desplazamos la imagen 16 pixeles a la izquierda */
a.icono3 {background-position: -32px 0px;} /* desplazamos la imagen 32 pixeles a la izquierda */

/* las propiedades de los íconos con efecto hover */
/* de manera similar, desplazamos la imagen a la izquierda y 16 pixeles hacia arriba */
a.icono1:hover {background-position: 0px -16px;}
a.icono2:hover {background-position: -16px -16px;}
a.icono3:hover {background-position: -32px -16px;}
```

Lo mismo podemos hacer *inline*, escribiendo el estilo en la etiqueta HTML. En este ejemplo usamos una imagen de 96x170 pixeles donde se combinan dos, una debajo de la otra:

En este ejemplo, usamos dos imágenes de 96x85 combinadas:

```
<div style="background: transparent url(URL_IMAGEN) no-repeat 0 0; height: 85px; width: 96px;" onmouseover="this.style.backgroundPosition='0px -85px';" onmouseout="this.style.backgroundPosition='0px 0px';"></div>
```

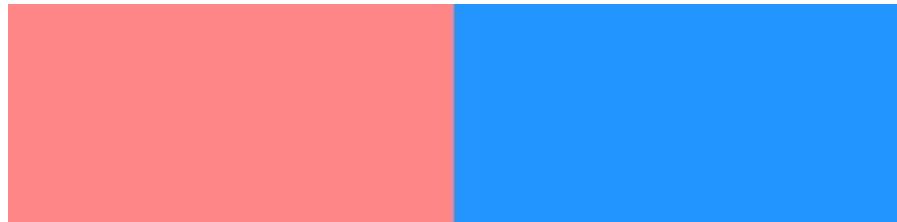
La posición se define en pixeles, primero la coordenada horizontal y luego la coordenada vertical. La posición 0px 0px es la esquina superior izquierda, y el desplazamiento lo calculamos sumando los anchos y/o los altos de cada parte y colocando ese valor como un número negativo.

¿Cuál es la ventaja? Fundamentalmente, cuando se trata de íconos con efectos *hover*, evitamos la demora que se produce cuando se carga la segunda imagen; al ser una sola, siempre está disponible y eso agiliza la visualización de las páginas.

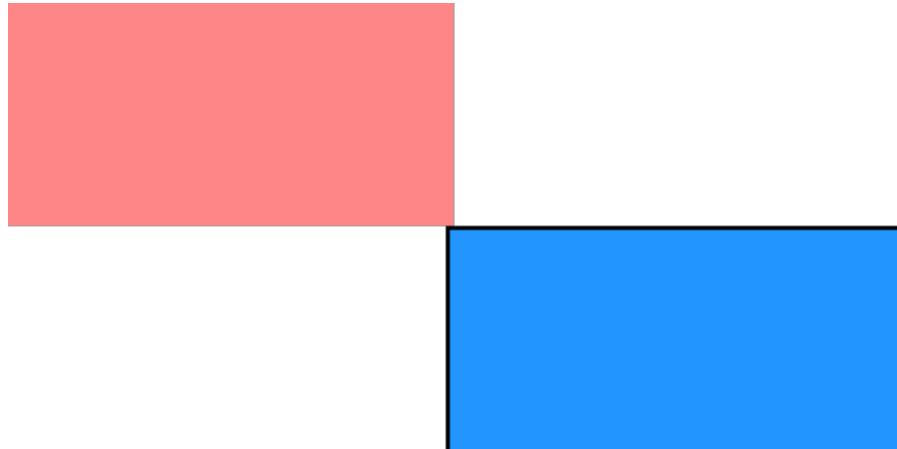
Obviamente, no es algo para aplicar en cualquier caso y siempre hay que tener en cuenta que el tamaño de las imágenes es un detalle a tener muy en cuenta cuando se trata de velocidades de carga pero, es un método efectivo y muy recomendable.

Los bordes ocupan espacio

El borde ocupa espacio por lo tanto, si el elemento tiene determinado ancho, cuando usamos esta propiedad, ese ancho se incrementará sin importar si hemos definido el ancho con `width` o no y eso crea problemas cuando se diagrama una página; por ejemplo, si tuviéramos un `<div>` de 400 pixeles de ancho y adentro colocamos otros dos de 200 pixeles cada uno y a esos les ponemos la propiedad `float`, se verán uno al lado del otro:



Pero si a uno de ellos le agregamos un borde, se irá para abajo porque no espacio:



Para solucionar esto, debemos aumentar el ancho del contenedor o reducir el ancho del `<div>` con el borde tantos pixeles como sea necesario.

Algo similar ocurre cuando se colocan bordes en elementos *inline* al utilizar estilos de `hover`, al colocar el cursor encima de este enlace, todo el texto se desplazará.

Esto, se soluciona colocándole un borde de color transparente al enlace y de ese modo, ya estamos “reservando” el espacio necesario:

```
a {border:3px solid transparent;}  
a:hover {border:3px solid #FFF;}
```

Pero `border` no es la única propiedad que nos permite recuadrar etiquetas, hay otra que no utilizamos mucho y que hace algo parecido, se trata de la propiedad `outline`.

Y decimos que es parecida porque tiene dos características particulares. La primera es que no nos permite definir cuál borde agregar, siempre se agregarán los cuatro; la segunda es que no ocupa espacio así que en el ejemplo anterior, si usáramos `outline` en lugar de `border`, el `<div>` no “bajaría”.

margin versus padding

Márgenes, *paddings*, flotaciones, todo da igual, usamos las propiedades de modo intuitivo, probando con una o con otra, agregando valores, disminuyéndolos o aumentándolos pero llega un momento en que todos esos pequeños ajustes se vuelven inmanejables.

En el CSS no hay propiedades malas y propiedades buenas, lo que deberíamos hacer para no complicarnos es entender las diferencias entre ellas porque a veces parecen ser los mismo pero no lo son.

La propiedad **margin**, separa TODA la etiqueta de la que está arriba y de la que está abajo; en cambio, la propiedad **padding**, separa el contenido de la etiqueta de sus propios bordes es decir, cambia su tamaño haciéndola más ancha o más alta.

Como esto suele aplicarse a elementos de bloque (etiquetas `<p>`, `<div>`, etc), a menos que lo indiquemos expresamente, el ancho que ocupará será el mismo que el rectángulo que la contiene así que si queremos darle un ancho diferente, debemos usar la propiedad **width**.

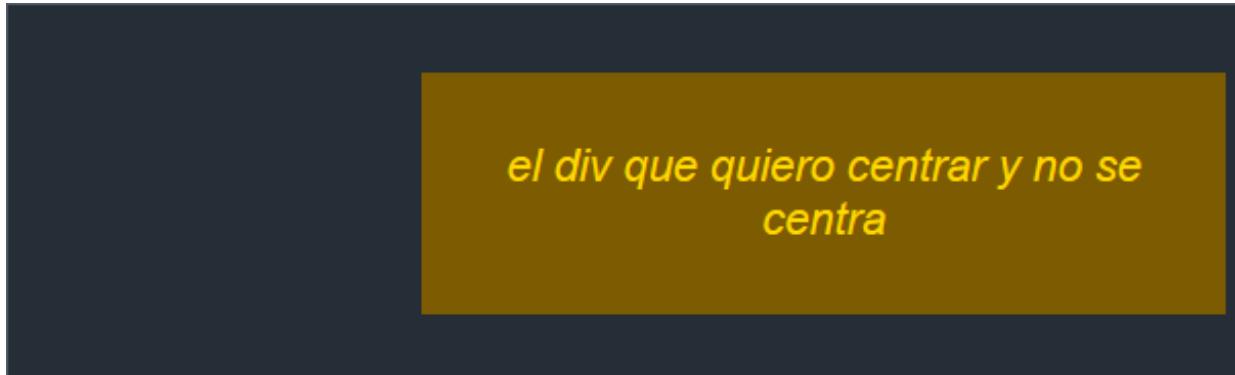
Si tiene un ancho definido, podemos centrarla usando los márgenes; para eso, establecemos el valor a derecha e izquierda con la palabra `auto`; por ejemplo:

```
margin: 0 auto; /* centrado sin margen superior e inferior */  
margin: 30px auto; /* centrado con el mismo margen superior e inferior */  
margin: 30px auto 10px; /* centrado con mismo margenes distintos */
```

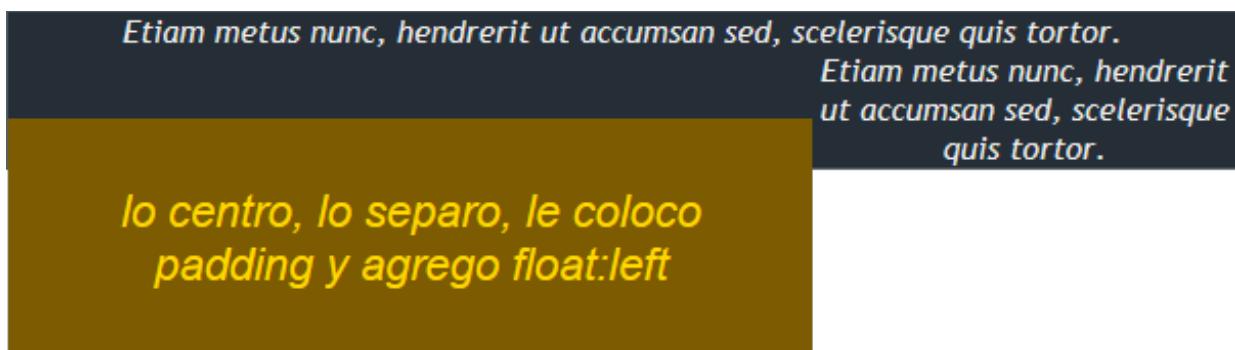
El problema que surge muchas veces al tratar de centrar algo, es que el elemento que queremos centrar, se encuentra dentro de otros y esos otros también tienen márgenes o *paddings* por lo tanto, todo se tergiversa y ahí, suele recurrirse a márgenes negativos o cosas raras cuando, en realidad, bastaría poner todo a cero y eliminar esos valores innecesarios:

```
<div>  
  <div style="padding-left: 80px;">  
    <div style="margin-left: 100px;">  
      <div style="margin:30px auto;padding:30px;width:300px; ;">  
        el div que quiero centrar y no se centra  
      </div>  
    </div>  
  </div>  
</div>
```

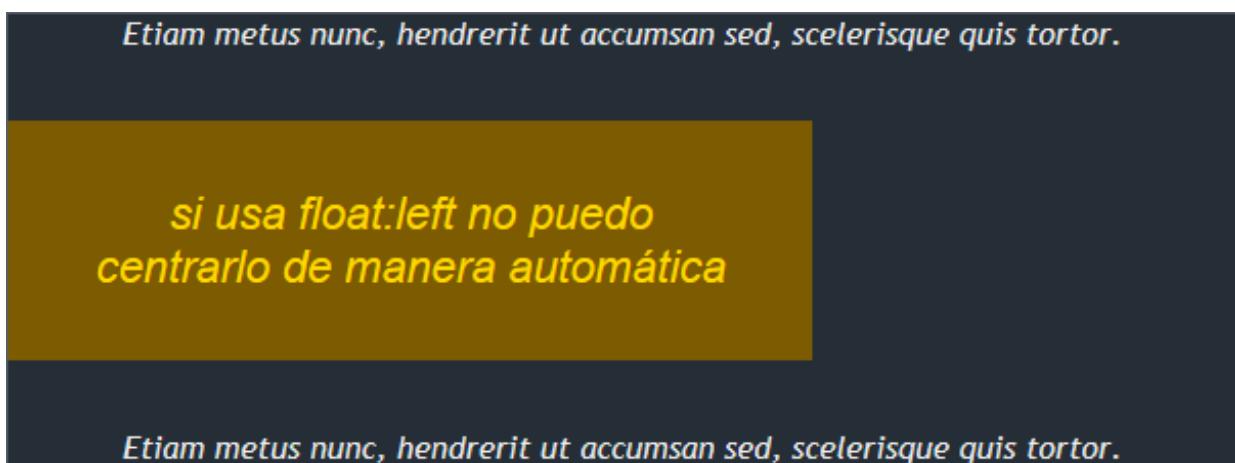
Quiero centrar el <div> pero no se centra:



Otro problema es el que surge cuando usamos flotaciones (propiedad **float**) que es muy útil pero tiene sus bemoles:



Como se ve en el ejemplo, el texto inferior se ha ido de paseo y se muestra arriba en lugar de abajo. No, no hay error, lo que hace **float** es exactamente eso, cambia el flujo normal de las etiquetas (una debajo de la otra) y si queremos que lo de abajo permanezca abajo, a esa última debemos agregarle la propiedad **clear:both**; para indicar que de ahí en adelante, todo vuelve a la normalidad:



Además, si usamos **float**, la palabra **auto** deja de tener efecto así que la etiqueta no se centrará y para hacerlo, no queda otro remedio que usar **margin-left**, calculando el valor con un poco de aritmética; algo así:

((width contenedor - width contenido) / 2) - padding contenido - border-width contenido

Etiam metus nunc, hendrerit ut accumsan sed, scelerisque quis tortor.

si usa float se debe centrar con margin

Etiam metus nunc, hendrerit ut accumsan sed, scelerisque quis tortor.

Otro ejemplo donde colocamos dos rectángulos, uno dentro de otro. El exterior tendrá un color de fondo y el interior otro:

 Lorem ipsum dolor sit
 amet, consectetur
 adipiscing elit.

```
<div style="width: 100%;">  
  <div style="padding: 30px; width: 170px;">  
    ... el texto o el contenido ...  
  </div>  
</div>
```

El ancho del rectángulo exterior es 100%, ocupa todo el ancho disponible, dependiendo de dónde esté, será más corto o más largo. El ancho del rectángulo interior es fijo y tiene un valor cualquiera; además tiene agregada la propiedad **padding** que hace que eso que está dentro de él, se separe de los cuatro bordes.

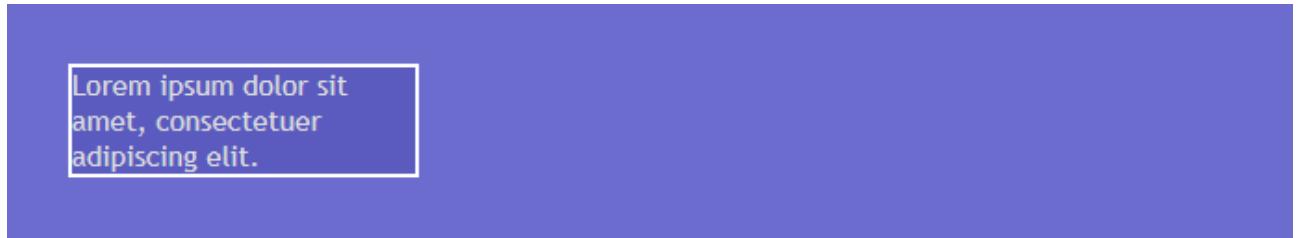
Si en lugar de **padding** usáramos **margin**, el resultado no sería el mismo:

 Lorem ipsum dolor sit
 amet, consectetur
 adipiscing elit.

```
<div style="width: 100%;>
  <div style="margin: 30px; width: 170px;">
    ... el texto o el contenido ...
  </div>
</div>
```

La diferencia entre ambas propiedades podría resumirse así: **margin** actúa hacia afuera, separa el rectángulo de aquel otro que lo contiene; **padding** actúa hacia adentro, separa el contenido de ese rectángulo de sus propios bordes.

Pero ¿no lo separa de abajo ni de arriba? Lo hace, pero no lo vemos. Para que el margen sea visible sobre los cuatro lados, el **<div>** contenedor debe “flotar”:



```
 Lorem ipsum dolor sit  
 amet, consectetuer  
 adipiscing elit.
```

```
<div style="float: left; width: 100%;>
  <div style="margin: 30px; width: 170px;">
    ... el texto o el contenido ...
  </div>
</div>
```

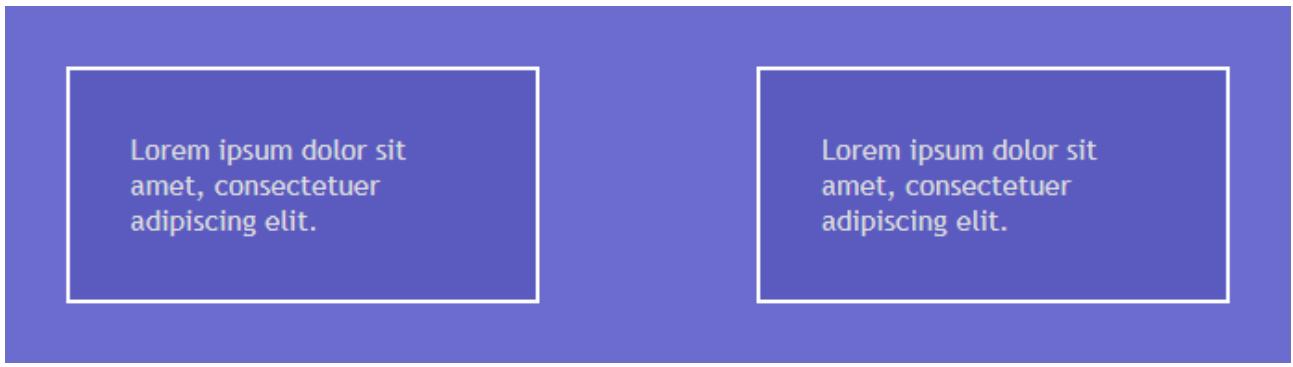
Entonces, si quisiera separar el rectángulo del contenedor y además, separar el contenido, debería usar ambas propiedades:



```
 Lorem ipsum dolor sit  
 amet, consectetuer  
 adipiscing elit.
```

```
<div style="float: left; width: 100%;>
  <div style="margin: 30px; padding: 30px; width: 170px;">
    ... el texto o el contenido ...
  </div>
</div>
```

Bien. Ahora podemos agregarle un segundo rectángulo y les ponemos la propiedad **float** a todos ellos así los vemos juntos.



```
<div style="float: left; width: 100%;">
  <div style="float: left; margin: 30px; padding: 30px; width: 170px;">
    ... el texto o el contenido ...
  </div>
  <div style="float: right; margin: 30px; padding: 30px; width: 170px;">
    ... el texto o el contenido ...
  </div>
</div>
```

Uno, flota hacia un lado y el otro flota hacia el otro.

Ahora vamos a tratar de verlo con un último ejemplo. Imaginemos que tenemos un rectángulo (un `<div>`) del que desconocemos su ancho (o varía) y queremos incluir en su interior varios rectángulos más con cierto contenido y que se muestren del mismo tamaño. Por ejemplo, que quede algo así:



Si a esos cuatro rectángulos les damos un ancho fijo y se ve bien, será un milagro ya que desconocemos los tamaños pero, nos queda la aritmética así que pensamos, bueno, dividamos el total en cuatro partes iguales y en lugar de colocar un valor absoluto para `width`, usemos porcentajes (25%).

Y tampoco funcionará porque uno de ellos se irá para abajo ¿Qué pasa? El ancho de `width` no incluye ni el `margin` ni el `padding`, así que se los saco. No es lo que quiero pero al menos ahora está dividido en partes iguales.

```
<div style="float: left; width: 100%;">
  <div style="float: left; width: 25%;"> ... </div>
  <div style="float: left; width: 25%;"> ... </div>
  <div style="float: left; width: 25%;"> ... </div>
  <div style="float: left; width: 25%;"> ... </div>
</div>
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

El problema es siempre el mismo, el ancho es el definido por:

width + border + padding + margin

con ciertas variantes según sea el navegador pero cada una de esas propiedades, hace que el rectángulo sea cada vez más grande así que hay que recurrir a la solución mágica para la mayoría de los problemas de este tipo. Si no podemos alinear un <div>, pongamos todo eso en otro.

Yo quiero que esos rectángulos interiores se separen entre si y que el contenido también se separe así que debo meterlos dentro de otros que son los que flotarán y que medirán el 25% del ancho total y por último, meteré todo eso en el primer rectángulo::

```
<div style="width: 100%; float: left;">
  <div style="float: left; width: 25%;">
    <div style="margin: 10px; padding: 10px;"> ... </div>
  </div>
  <div style="float: left; width: 25%;">
    <div style="margin: 10px; padding: 10px;"> ... </div>
  </div>
  <div style="float: left; width: 25%;">
    <div style="margin: 10px; padding: 10px;"> ... </div>
  </div>
  <div style="float: left; width: 25%;">
    <div style="margin: 10px; padding: 10px;"> ... </div>
  </div>
</div>
```

Lore ipsum dolor
sit amet,
consectetuer
adipiscing elit.

Si quiero distribuirlos de manera pareja, en lugar de colocar en todos ellos un margen igual para los cuatro lados, coloco un margen derecho igual a 0 en los tres primeros:

margin: 10px 0 10px 10px;

Alinear verticalmente

Alinear algo verticalmente no suele parecer tan evidente. No hay una solución universal para alinear cosas, todo depende. Lo más simple es centrar verticalmente los textos de una etiqueta. En cualquier bloque, si establecemos una altura con **height**, basta colocar la propiedad **line-height** con el mismo valor para que el texto quede centrado:

```
<div style="height: 100px; font-size: 20px; line-height: 100px;">  
    ... cualquier texto ...  
</div>
```

Valores inferiores a la altura (**line-height: 50px;**) harán que el texto se muestre arriba y esto es lo que ocurre normalmente ya que el valor estándar de esa propiedad es **normal** y eso significa que es más o menos igual a una vez y media la altura de la fuente del texto.

Valores superiores a la altura (**line-height: 150px;**) harán que el texto se muestre abajo

Para centrar elementos que tienen diferentes alturas, lo más común es usar la propiedad **vertical-align** con un valor de **middle**. Por ejemplo:

```
<div style="text-align: center; vertical-align: middle;">  
    <span style="font-size: 22px;">un texto</span>  
    <span style="font-size: 32px;">un texto</span>  
    <span style="font-size: 16px;">un texto</span>  
</div>
```

Pero esto no centrará demasiado bien, la propiedad, deberíamos colocarla en cada una de las etiquetas internas y no en el contenedor:

```
<div style="text-align: center;">  
    <span style="vertical-align: middle; font-size: 22px;">un texto</span>  
    <span style="vertical-align: middle; font-size: 32px;">un texto</span>  
    <span style="vertical-align: middle; font-size: 16px;">un texto</span>  
</div>
```

Esto, es aplicable tanto a textos como a imágenes:

```
<div style="font-size: 16px; text-align: center;">  
    un texto cualquiera  
      
    otro texto cualquiera  
</div>
```

Si las cosas son más complicadas, nada impide usar una **<table>** o la propiedad **display** con valores **table** y **table-cell**.

```
<style>
  .contenedor {
    display:table;
  }
  .contenido {
    display:table-cell; /* esta es la propiedad clave */
    text-align:center; /* esto centra el contenido horizontalmente */
    vertical-align:middle; /* esto centra el contenido verticalmente */
  }
</style>
```

```
<div class="contenedor">
  <div class="contenido">cualquier contenido</div>
  <div class="contenido">cualquier contenido</div>
  <div class="contenido">cualquier contenido</div>
</div>
```

Y todo se irá alineando sólo, sin necesidad de saber previamente cuales son los tamaños.

Sobre IDs y bloques ocultos

Cuando insertamos un bloque con cierto contenido, utilizamos una etiqueta. Dependiendo de cómo lo hacemos o el uso que le daremos, estará en una etiqueta `<div>`, `<p>`, ``, etc. Sin importar cuál sea, a los efectos prácticos, esa etiqueta no es otra cosa que un rectángulo donde adentro “hay algo” que puede ser cualquier cosa, incluyendo otras etiquetas.

Esos rectángulos pueden ser controlados de manera individual si les colocamos un nombre único, es decir si los identificamos con precisión y para eso, usamos el atributo `id`. Por ejemplo:

```
<div id="contenedor"> .....
```

El nombre que le ponemos no importa, puede ser cualquiera y contener letras, números o guiones pero no espacios en blanco. Puede estar en mayúsculas, minúsculas o ambas cosas aunque debemos recordar que JavaScript es sensible a eso así que cuando usamos esos nombres en un código, hay que usar el mismo criterio ya que `id="TEXTO1234"` e `id="texto1234"` no serán el mismo.

Los `id` deben ser únicos, es decir, no debe haber dos iguales en una misma página porque si luego los queremos manipular, el navegador no sabrá a cuál nos referimos.

Es gracias a esos `id` que podemos usar JavaScript para cambiar dinámicamente (cuando la página ya se ha cargado) las propiedades de esos bloques; por ejemplo, mostrarlos u ocultarlos a voluntad.

Uno podría preguntarse para qué querríamos tener elementos ocultos dentro de una página web pero, esto es algo muy utilizado ya que nos permite reducir el espacio gráfico, permitir que el usuario seleccione el tipo de información que quiere ver, etc.

Usando CSS es sencillo ocultar el contenido de algo, basta colocarle la propiedad `display: none`:

```
<div style="display: none;">este es el contenido</div>
```

Para hacerlo visible, usamos la propiedad contraria y que es la propiedad por defecto de la mayoría de las etiquetas así que:

```
<div style="display: block;">este es el contenido</div>
```

Esa propiedad es común a todas las etiquetas así que, podemos ocultar cualquier parte de una página web. Muchos creen que ocultar algo utilizando este método alivia la carga de una página pero eso no es cierto.

Justamente, el que eso no sea cierto es lo que nos permite hacer el proceso inverso, es decir, mostrarlo, sin necesidad de recargar nada y por lo tanto, el efecto es casi instantáneo por una sencilla razón, ya está allí desde el inicio, ya se cargó junto con el resto.

Obviamente, si algo está oculto siempre debe haber algún otro elemento que nos permita mostrarlo pero, ¿cómo hacemos para permutar entre ambos estados? ¿cómo hacemos para que algo oculto se muestre y algo visible se oculte? Para eso usamos JavaScript, dándole una instrucción que cambie el estilo CSS de algún elemento al que le hemos dado un nombre. En general, sería así:

```
el_ID.style.propiedad = "valor";
```

Para que no tengamos problemas a la hora de indicarle al navegador cuál es la etiqueta que vamos a modificar, en lugar de usar el nombre *elID*, usamos una función interna:

```
getElementById("el_ID").propiedad = "valor";
```

o bien:

```
document.getElementById("el_ID").propiedad = "valor";
```

La forma de identificar la propiedad suele ser similar a la propiedad CSS aunque hay variaciones. En este caso serán:

```
getElementById("el_ID").style.display = "block";
getElementById("el_ID").style.display = "none";
```

¿Cómo aplicamos esto? Podríamos usar enlaces donde agregamos el atributo onclick:

```
<a href="#" onclick="getElementById('ejemplo').style.display='block';">click para ver</a>
<div id="ejemplo" style="display:none;>
    el contenido que mantenemos oculto
</div>
```

¿Y cómo lo ocultamos otra vez? Una forma sería agregarle otro enlace dentro del elemento oculto que lo cierre:

```
<a href="#" onclick="getElementById('ejemplo').style.display='block';">click para ver</a>
<div id="ejemplo" style="display:none;>
    el contenido que mantenemos oculto
        <a href="#" onclick="getElementById('ejemplo').style.display='none';">ocultar</a>
</div>
```

Funciona pero podemos hacerlo mejor y simplificarlo. Por ejemplo, es muy común usar una función que nos permite permutar entre dos estados. A esto, suele llamarse efecto *toggle*:

```
function toggle(objID){  
    var o = document.getElementById(objID);  
    if(o.style.display=="block") {  
        o.style.display = "none";  
    } else {  
        o.style.display = "block";  
    }  
}
```

Y lo aplicamos de este modo:

```
<a href="#" onclick="toggle('ejemplo');">click para ver</a>  
<div id="ejemplo" style="display:none;">  
    el contenido que mantenemos oculto  
</div>
```

Las posiciones absolutas son relativas a “algo”

Cuando diseñamos una página web o simplemente agregamos etiquetas, estas se acomodan de manera natural, ya sea una al lado de la otra o bien una debajo de la otra, dependiendo del tipo que sean. El contenido fluye naturalmente, de arriba hacia abajo y de izquierda a derecha, tal como ocurre cuando escribimos o leemos.

Para modificar esa secuencia usamos distintas propiedades de CSS; les colocamos márgenes, cambiamos su tipo con **display** o las hacemos flotar.

De un tiempo a esta parte, se está empleando una nueva forma de posicionar “cosas” en la página, utilizando para ello la propiedad respectiva que se llama **position** y, que como toda propiedad de CSS, tiene un valor por defecto aunque no la definamos. Toda etiqueta tiene el valor **static** que no hace nada salvo decirle al navegador que esa etiqueta es “normal”.

Las variantes **relative** y **absolute** que admiten esa propiedad se suelen usar sin tener en cuenta qué son o que hacen y por lo tanto, cómo afectan al resto de las etiquetas.

Usar posiciones absolutas para ubicar un elemento de manera precisa en una página web es uno de los métodos más cómodos porque requiere propiedades que cualquier navegador entiende y nos evita agregar márgenes, *paddings* y flotaciones que siempre perturban ya que afectan al resto de las etiquetas.

Es sencillo, basta agregarle a ese elemento la propiedad **position:absolute** y luego, establecer los valores de **top**, **right**, **bottom** y/o **left** teniendo en cuenta que:

top:0; left:0; es el ángulo superior izquierdo
top:0; right:0; es el ángulo superior derecho
bottom:0; left:0; es el ángulo inferior izquierdo
bottom:0; right:0; es el ángulo inferior derecho

Es decir, los cuatro extremos de ¿qué? Esa es la clave; tener la respuesta es lo que evita los problemas.

Lo primero que debe tenerse en claro es que “lo absoluto” no existe y que las posiciones absolutas no son absolutas en abstracto, lo son, con respecto a algo: a un contenedor.

Toda etiqueta está dentro de un contenedor, es decir, dentro de otra etiqueta; por ejemplo, esta imagen está dentro de una etiqueta **<div>** (su contenedor) que a su vez, está dentro de un **<div>** que es el contenedor de ambas, que está dentro de otro **<div>** que es el contenedor de las tres.

Si colocamos **position:absolute** en una etiqueta cualquiera, el navegador la posicionará buscando “hacia atrás” la primera etiqueta que NO TENGA la propiedad **position:static**; y si no hay ninguna, tomará como referencia el **<body>**, es decir, la esquina superior izquierda de la ventana del navegador.

```

<body>
.....
<div>
  <div>
    <div>
      
    </div>
  </div>
</div>
.....
</body>

```

Si agregáramos **position:absolute** a la etiqueta **** ¿dónde se vería?

```

```

No hay forma de adivinarlo, no depende de las propiedades de la imagen sino de las propiedades de su contenedor ya que es este el que define cuáles son los “límites” del rectángulo, cuáles son las coordenadas 0:0 y lo elemental es que definamos eso agregándole a ese contenedor, la propiedad **position:relative**.

Este es el error más común, suponer que si algo tiene una posición absoluta, el navegador entenderá dónde queremos que se vea pero, los navegadores no piensan, sólo obedecen órdenes.

Tampoco es cierto que agregando esa propiedad, todo se resuelva porque por si sola no causa mayor efecto.

Poder usar estas propiedades para ubicar algo es lo que las hace una forma sencilla de diseñar algo pero, si podemos posicionar un objeto de ese modo ¿para qué agregar otras propiedades como **margin** y **float**? Esto también es bastante común y debería evitarse. Si posicionamos algo de modo absoluto, los márgenes suelen ser innecesarios y las flotaciones, inútiles. Así que ¿para qué complicarse la vida?

Lo mismo puede decirse se **display**; salvo muy raras excepciones, es innecesario agregar esa propiedad en la regla.

Por supuesto, como la posición es absoluta, **margin: 0 auto;** no centra absolutamente nada; mucho menos lo hará **text-align**. Si queremos centrar algo que tenga este tipo de propiedad, debemos conocer su ancho. Por ejemplo, en este caso, un rectángulo de 200x120 lo centramos usando **left** y un margen izquierdo negativo; y con **top** lo podemos centrar verticalmente:

```
#ejemplo {
  left: 50%;
  margin-left: -100px;
  margin-top: -60px;
  position: absolute;
  top: 50%;
}
```

Los pixeles son unidades indivisibles

Es muy común ver que en las reglas de estilo se usen valores en pixeles con decimales; esto no es algo que vaya a provocar un error pero es inútil ya que los pixeles son unidades indivisibles.

Así que si algo mide 200 pixeles, 200.5 o 200.8 es indistinto pero ... alguien puede decir “no, si yo pongo un decimal se ve más grande que si no lo pongo” y eso es verdad pero no es cierto.

Lo que ocurre es que los navegadores interpretan esos decimales de distintas formas. En términos generales:

1. en Firefox los decimales superiores a .5 serán redondeados hacia arriba
2. en IE los decimales serán redondeados hacia arriba
3. en Chrome el decimal será ignorado

Obviamente, son sutilezas pero, un pixel de diferencia puede significar mucho ya que si colocamos dos elementos uno al lado del otro y uno de ellos supera el ancho del contenedor, se mostrará debajo.

Un ejemplo; tengo dos contenidos que flotan y miden 200.5 pixeles; hago cuentas y llego a la conclusión que necesito un contenedor de 401 pixeles ... es fácil pero en Firefox e IE no se verán uno al lado del otro y en Chrome si.

Moraleja: Para evitar un problema, lo mejor es no generarlo así que, simplemente, no debemos usar valores decimales y listo ¿Para qué complicarse la vida?

Colocar texto alrededor de una imagen

Eso de centrar una imagen y ponerle el texto alrededor no tiene una solución universal.

Una posibilidad es utilizar columnas donde tengamos una imagen centrada y dos textos, uno a su derecha y otro a su izquierda. La primera solución para eso es algo así:

```
<style>
#contenedor-center {
    margin: 0 auto;
    text-align: center;
    width: 620px;
}
#texto-L, #texto-R {width: 200px;}
#texto-L {float: left; text-align:right;}
#texto-R {float: right; text-align:left;}
</style>

<div id="contenedor-center">
    <div id="texto-L">
        <p> el texto a mostrar </p>
    </div>
    <div id="texto-R">
        <p> el texto a mostrar </p>
    </div>
    
</div>
```

Cras tincidunt auctor metus in interdum. Nam semper varius est, sed pulvinar tellus dignissim vel. Fusce lobortis congue interdum. Morbi lobortis gravida rutrum. Fusce scelerisque fringilla diam, vitae viverra justo rhoncus auctor. Nullam metus purus; sollicitudin sit amet pulvinar ut, euismod at odio. Duis bibendum lacus nec velit posuere pulvinar. Praesent tristique elit a felis tincidunt eu dictum diam ultrices. Donec accumsan posuere accumsan? Maecenas condimentum vulputate ante ac egestas. Suspendisse et enim est, interdum dapibus metus. Nullam venenatis molestie rutrum.



Aenean sollicitudin urna quis nibh ultricies rutrum. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aenean nec commodo ligula. Ut eleifend, justo elementum tempor pharetra, eros nunc rhoncus nibh, nec eleifend tellus mi vitae mauris. Curabitur a mauris varius est bibendum iaculis? Nam in felis mauris. Cras odio dui, fermentum ac tincidunt in; condimentum non nulla. Proin convallis; lectus sit amet sollicitudin rutrum, sapien quam porta purus, et feugiat felis lacus id dui. Cras sed pretium quam.

Se verán tres columnas pero el texto no rodeará la imagen; abajo quedará un espacio en blanco si es que ese texto es largo.

Una solución alternativa es utilizar el pseudo-elemento ::before para crear un “agujero” y allí poner la imagen:

```
<style>
    #contenedor-center {
        margin: 0 auto;
        position: relative;
        width: 620px;
    }
    #imagen-center {
        left:50%;
        margin-left:-100px; /* el margen es la mitad del ancho de la imagen */
        position: absolute;
        top: 0;
    }
    #texto-L, #texto-R {width: 49%;}
    #texto-L {float: left;}
    #texto-R {float: right;}
    #texto-L:before, #texto-R:before {
        content: "";
        height: 300px; /* la altura es igual a la altura de la imagen */
        width: 100px; /* el ancho es la mitad del ancho de la imagen */
    }
    #texto-L:before {float: right;}
    #texto-R:before {float: left;}
</style>

<div id="contenedor-center">
    
    <div id="texto-L">
        <p> el texto a mostrar </p>
    </div>
    <div id="texto-R">
        <p> el texto a mostrar </p>
    </div>
</div>
```

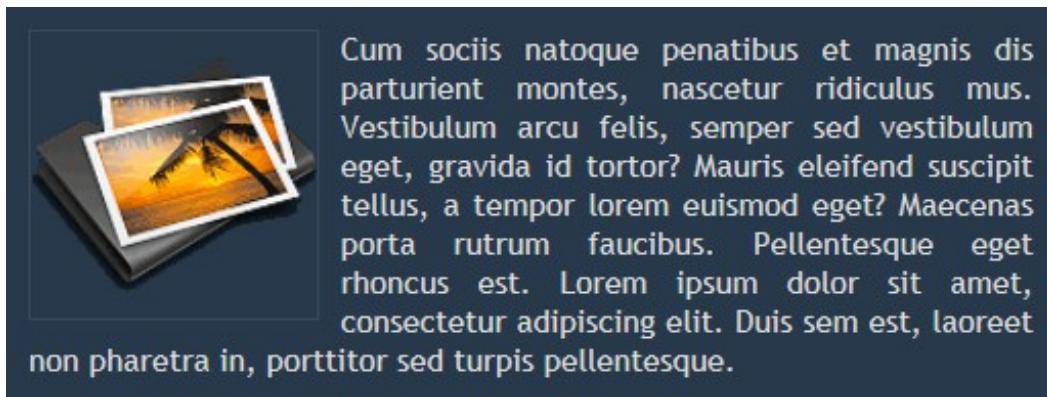
Ajustar el tamaño de una imagen al texto

Por lo general, usando CSS, armar un texto combinado con una imagen es bastante sencillo aunque a veces, ajustar ambas cosas se vuelve una tarea titánica.

Supongamos que tenemos dos imágenes, una de 128x128 y otra de 256x256 y que queremos usarlas como “adorno” de un texto.

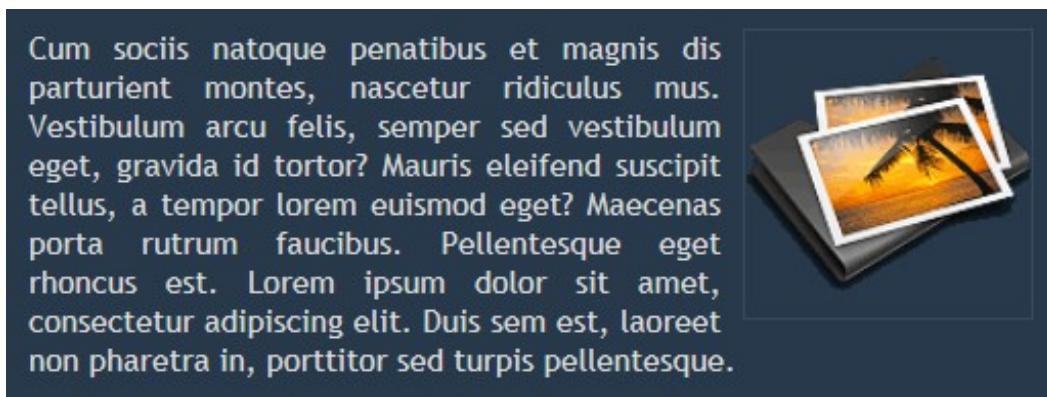
Usando la propiedad **float**, podemos colocar el texto “rodeando” la imagen, ya sea a un lado o al otro.

```
<div style="text-align: justify;">
  
    el texto a mostrar
</div>
```

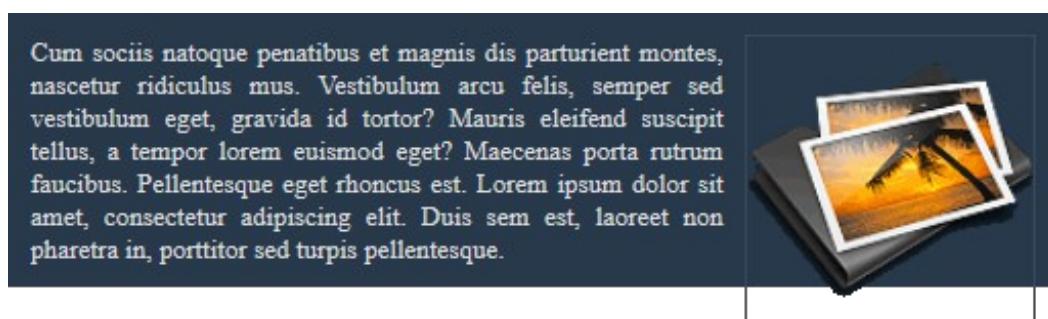


Lo mismo hacemos para el otro lado:

```
<div style="text-align: justify;">
  
    el texto a mostrar
</div>
```



Sólo invertimos los códigos, uno flota a la izquierda y tiene un margen a la derecha, el otro flota a la derecha así que tiene un margen a la izquierda pero ¿qué pasa si por cualquier motivo cambiamos el tamaño o el tipo de fuente del texto?



Todo se descabla y claro, deberíamos redimensionar la imagen para mantener el efecto. Eso lo podríamos hacer usando los atributos **width** y **height**.

Basta colocar el ancho y el navegador se encargará de calcular el alto de forma automática para mantener la proporción. O colocar sólo el alto. O colocar ambos valores y cambiar esa proporción. Si la original era de 256x256 la cambiamos a 100x100:

```





```

Sea como sea, deberíamos “recalcular” el tamaño de manera individual ... sigue siendo mucho trabajo.

Sin embargo, hay una técnica que nos facilita hacer esto de manera genérica, usando CSS. Para ello, en lugar de utilizar pixeles para dimensionar las imágenes, utilizaremos la unidad **em** lo que significa que el tamaño de la imagen tendrá una relación con el tipo de fuente del texto.

¿Cómo calculamos eso? No hay otra forma que usar la aritmética y una calculadora.

Supongamos que estoy satisfecho con el modelo creado. Conozco dos datos, sé que la imagen debe tener 128x128 y que la fuente del texto tiene 14 pixeles. Uso una fórmula de conversión:

$$(1 / \text{TamañoFuente}) * \text{AncholImagen} = \text{AncholImagen en unidades em}$$
$$(1 / \text{TamañoFuente}) * \text{AltolImagen} = \text{AltolImagen en unidades em}$$

En este caso:

$$(1 / 14) * 128 = 9.14$$

Así que puedo cambiar el código a usar.

```
<div style="text-align: justify;">
    
    el texto a mostrar
</div>
```



Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Vestibulum arcu felis, semper sed vestibulum eget, gravida id tortor? Mauris eleifend suscipit tellus, a tempor lorem euismod eget? Maecenas porta rutrum faucibus. Pellentesque eget rhoncus est. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sem est, laoreet non pharetra in, porttitor sed turpis pellentesque.

De esta manera, el texto y la imagen mantendrán una relación pre-establecida, sin importar ni el tipo de fuente ni el tamaño de la imagen que usemos.

Flotaciones, fondos, problemas, alternativas

En una página web, todo es relativo, casi no existen reglas absolutas que funcionen en el 100% de los casos y eso, pese a que muchos opinan lo contrario, no es malo; es lo que permite la variedad aunque también genera los problemas.

Uno de esos problemas suele darlo la propiedad **float** que es útil pero debe ser usada con prudencia (como todo) y sólo ahí donde sea necesario ya que “trastorna” la forma en que se genera una página, quitando ese contenido del orden natural que es el mismo que se utiliza para leer o escribir (de arriba hacia abajo y de izquierda a derecha).

Frente a cosas raras, lo primero que debería preguntarme es si la propiedad es necesaria o no pero, supongamos que si y tenemos dos **<div>** que flotan a la derecha y un texto:

```
<div class="demoflotante"> ..... </div>
<div class="demoflotante"> ..... </div>
<div> un texto </div>
```

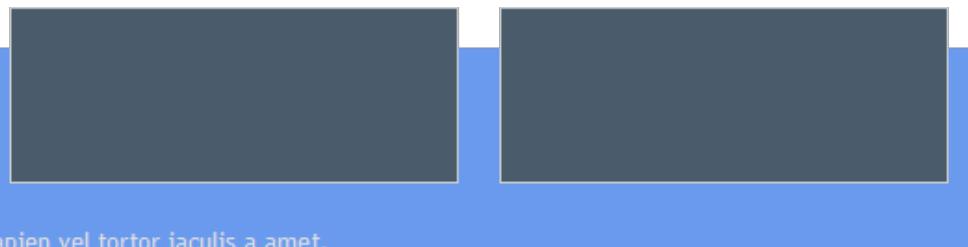
Si esos **<div>** tienen un ancho pequeño, veríamos algo como esto:

Proin fringilla ultricies
diam, id sodales ipsum
suscipit sit amet. Vivamus
in libero elit, eu cursus
magna. Maecenas sagittis
mollis sapien, at hendrerit
dui viverra a! Nulla rutrum sapien vel tortor iaculis a amet.



El texto, “rodea” a los dos **<div>** flotantes pero ... ocupa un rectángulo mayor ¿Cómo puede verse ese espacio? por ejemplo, colocando un color de fondo:

Proin fringilla ultricies
diam, id sodales ipsum
suscipit sit amet. Vivamus
in libero elit, eu cursus
magna. Maecenas sagittis
mollis sapien, at hendrerit
dui viverra a! Nulla rutrum sapien vel tortor iaculis a amet.



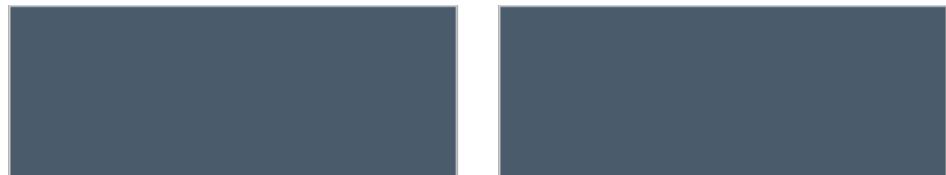
Tal como hemos creado el HTML, da la impresión que quisiéramos que el texto se viera debajo y ahí hay tres soluciones posibles básicas.

La primera es agregarle al **<div>** del texto, la propiedad **clear:both** para eliminar las flotaciones previas y volver al orden natural o **clear:right** ya que los superiores flotan a la derecha:



Proin fringilla ultricies diam, id sodales ipsum suscipit sit amet. Vivamus in libero elit, eu cursus magna. Maecenas sagittis mollis sapien, at hendrerit dui viverra a! Nulla rutrum sapien vel tortor iaculis a amet.

¿Qué pasará ahora con el rectángulo? Quedará perfectamente ubicado porque **clear** hace justamente eso, restaura el orden:



Proin fringilla ultricies diam, id sodales ipsum suscipit sit amet. Vivamus in libero elit, eu cursus magna. Maecenas sagittis mollis sapien, at hendrerit dui viverra a! Nulla rutrum sapien vel tortor iaculis a amet.

¿Cuál es el problema de este método? Si el texto tiene un margen superior, este no tendrá ninguna utilidad, será ignorado. En el ejemplo tiene un **margin-top: 100px** y como se ve, ni se inmuta.

El segundo método es incluir un **<div>** extra antes del texto:

```
<div class="demoflotante"> .....
```

Dependiendo de las propiedades de nuestra página, tal vez deberíamos agregar más propiedades para resetearlas todas ya que es posible que tengamos reglas de estilo que establezcan fondos, bordes o cualquier otra cosa y lo único que queremos es hacer un corte así que deberíamos chequear **font-size**, **line-height**, **margin**, **padding**, **border**, **background**, etc o intentar poner **height: 0** y ver si eso resuelve las cosas.

Hay un método más “moderno” que consiste en crear una clase con una regla genérica que se puede aplicar a cualquier etiqueta y suele llamar **clearfix**:

```
.clearfix:after, .clearfix:before {  
    clear: both;  
    content: "x";  
    display: block;  
    height: 0;  
    visibility: hidden;  
}
```

177

Esa clase la agregamos al <div> del texto:

```
<div class="demoflotante"> ..... </div>
<div class="demoflotante"> ..... </div>
<div class="clearfix">un texto</div>
```



Proin fringilla ultricies diam, id sodales ipsum suscipit sit amet. Vivamus in libero elit, eu cursus magna. Maecenas sagittis mollis sapien, at hendrerit dui viverra a! Nulla rutrum sapien vel tortor iaculis a amet.

El resultado final será similar a la primera opción, los márgenes verticales funcionan de manera extraña y el rectángulo tampoco queda definido tal como uno deseaba.

¿Cuál es la mejor solución?

Ninguna de ellas y todas. Depende. Basta saber qué hace cada una de ellas y listo, el resto, depende de nosotros.

Optimizar el CSS

¿Quién no quiere optimizar las cosas? Es obvio que siempre es mejor que algo funcione de manera óptima a que funcione de manera mediocre pero en la web, solemos leer demasiados consejos y más allá de las buenas intenciones de quienes los proponen, no todos son aplicables e incluso, algunos son discutibles. Por suerte, como en todo, siempre hay más de una opinión y más de una solución y cuando hablamos de herramientas, no hay ninguna que sea perfecta ni sirva para todo así que usar el sentido común es siempre la mejor opción aunque ya se sabe que ese es el menos común de los sentidos.

Hay algo que parece básico, desde el punto de vista del tiempo de carga, salvo excepciones, comprimir el CSS no es relevante y para quienes recién comienzan puede transformar el código en una maraña inmanejable que asusta e impide avanzar; el resultado, no será otra cosa que un montón de caracteres difíciles de digerir y difíciles de editar:

```
body{background-color:#343F4A;color:#CCC;font-family:'Trebuchet MS', Trebuchet, Helvetica, Arial, Verdana, Sans-Serif;font-size:12px;margin:0;min-width:920px;padding:0;text-align:center}.clear{clear:both;line-height:0;height:0}#navbar-iframe{height:0;visibility:hidden;display:none}a,a:visited,a:link{color:#AAA;outline:none;text-decoration:underline}a:hover{color:#FFF;outline:none}a img{border:none;outline:none;text-decoration:none}object{outline:none}h1,h2,h3,h4,h5,h6{font-family:'Trebuchet Ms', Helvetica, Arial, sans-serif;margin:10px 0}
```

Cada cuál deberá escoger la forma que le resulte más cómoda para escribir y organizar las cosas. Habrá quien use espacios, quien agregue tabulaciones, quien ordene las propiedades por tipo o alfabéticamente. Todo es aceptable y no hay que limitarse porque la clave es que encontremos lo que buscamos, que podamos editarla y que funcione.

La estética del código no es relevante en absoluto, lo que interesa es que sea claro para nosotros mismos y evitemos los errores más comunes.

TODAS las propiedades terminan con un punto y coma; es cierto que no es necesario agregar ese carácter en la última propiedad de una regla pero, mejor lo ponemos en todas para acostumbrarnos y evitar problemas.

```
a { /* esto es un error */
    color: #AAA
    outline: none
    text-decoration:underline
}

a { /* esto está bien aunque la última no tenga punto y coma */
    color: #AAA;
    outline: none;
    text-decoration: underline
}
```

```
a { /* esto está bien */
  color: #AAA;
  outline: none;
  text-decoration: underline;
}
```

Cuando una propiedad está formada por varias palabras, debe haber un carácter espacio entre ellos pero, no es necesario un espacio entre el carácter dos puntos y la primera palabra:

```
div {
  /* esto es un error porque entre el cierre del paréntesis
  y la palabra no-repeat falta un espacio */
  background: #AAA url()no-repeat left top;
}

div {
  /* esto está bien aunque no haya espacio entre los dos puntos y el color */
  background:#AAA url() no-repeat left top;
}

div {
  /* esto está bien */
  background: #AAA url() no-repeat left top;
}
```

Dentro de una etiqueta `<style>` JAMÁS se agregan otras etiquetas, sólo se agregan reglas de estilo con sus propiedades:

```
div {
  /* esto es un error */
  <!-- comentario comentario comentario --
  background: #AAA url()no-repeat left top;
}

div {
  /* esto está bien */
  /* comentario comentario comentario */
  background:#AAA url() no-repeat left top;
}
```

El CSS es fundamental, mucho más cuando se trata de páginas dinámicas ya que es lo que nos permite simplificar la forma en que las diseñamos. Los editores, nos dan herramientas para eso y esas herramientas son útiles pero a la vez son engañosas así que hay que utilizarlas con prudencia.

Es cierto que podemos formatear nuestros textos con un simple *click* y listo; ponemos el color de los párrafos, el tipo de fuente; todo genial pero ... ¿y si mañana cambiamos de idea?

Agregar estilos *inline*, es decir, con un atributo **style** dentro de la etiqueta, debe ser siempre el último recurso, nunca el primero.

Siempre que sea posible debemos establecer reglas generales; el color, las fuentes, todo eso que vamos a utilizar habitualmente y colocar esas reglas y propiedades juntas.

Si no queremos que los enlaces se subrayen establecemos la regla para todo el sitio y eventualmente, cuando queremos que alguno de ellos se subraye, lo indicamos expresamente; ese tipo de detalles hará que no debamos estar repitiendo estilos en cada etiqueta.

Sin embargo, hay otras optimizaciones posibles.

Eliminar los espacios innecesarios depende de nuestra forma de trabajo. Lo ideal es que no haya ninguno así que deberemos encontrar alguna solución intermedia que nos permita tener código reducido pero legible.

Optimizar los valores de los colores es otra, podemos cambiar los valores largos tipo #AAAAAA por cortos como #AAA o los valores tipo rgb() por su valor hexadecimal.

si estamos en la sutileza, colocar red en lugar de #FF0000 y, en términos generales, elegir siempre el valor que ocupe menos caracteres. Todos estos valores significan lo mismo:

#FFEBBC = BlanchedAlmond = rgb(100%,92%,80%) = rgb(255,235,205)

Podemos eliminar las unidades de longitud iguales a cero:

margin: 0px 10px

equivale a:

margin:0 10px

Unificar las definiciones cuando sea posible (border, background, font, margin, padding):

p {margin-bottom: 0px; margin-left: 100px; margin-right: 20px; margin-top: 0px;}

equivale a:

.p {margin:0 20px 0 100px;}

Lo mismo ocurre con la propiedad **background**:

p {background-attachment: scroll; background-color: transparent; background-image: url(URL_imagen); background-position: left 50%; background-repeat: no-repeat;}

equivale a:

p {background: transparent url(URL_imagen) no-repeat scroll left 50%;}

Sería bueno revisar que no hubiera definiciones repetidas y en ese caso, unificarlas (es algo bastante común); lo mismo ocurre con propiedades que se sobrescriben:

```
a, a:visited, a:link {  
    color:#AAA;  
    outline:none;  
    text-decoration:underline;  
    color:#CCC;  
}
```

Consejos para trabajar con CSS

1. Separar el CSS, ya sea por estructura, colores o secciones.

```
div.contMenu{height:40px; width:300px;} /* estructura */  
div.contMenu{border-color:#999999;} /* colores */
```

2. Agrupar los estilos.

```
/* layout  
----- */  
/* tipografía  
----- */  
  
/* generales */  
body{}  
h1{}  
h2{}  
p{}  
  
/* clases */  
.clase1{}  
.clase2{}
```

3. Comentar el código. Si hay que cambiar algo todo se hace más fácil.

4. Agrupar propiedades. En lugar de usar varias propiedades, usar la propiedad general.

```
body{  
    background: #FFFFFF url(..../img/bglmg.gif) no-repeat fixed center center;  
}
```

en lugar de:

```
body{  
    background-color: #FFFFFF;  
    background-image: url(..../img/bglmg.gif);  
    background-repeat: no-repeat;  
    background-attachment: fixed;  
    background-position: center center;  
}
```

5. Utilizar mayúsculas al codificar colores.

No usar los nombres de los colores (aqua, black, blue, fucshia) ni abreviarlos, usar siempre los 6 caracteres #RRGGBB.

6. Usar minúsculas para definir los nombres de las clases.

7. Cerrar todas las declaraciones con punto y coma, aunque sea la última.

8. Usar tabulaciones con una propiedad por línea.

9. Usar selectores descendentes para mantener los estilos agrupados.

```
#header {}  
#header .logo {}  
#header .logo img {}
```

10. No perder de vista los DIV. Añadir un comentario tras las etiquetas de apertura y de cierre que informe de que se trata.

11. Utilizar ID sólo para definir grandes bloques. Tienen prioridad sobre las clases y por lo tanto bloquean algunas definiciones individuales.

12. Hay que tratar de definir siempre siguiendo un mismo orden. Por ejemplo: posición, dimensiones, márgenes, tipografía, fondos y bordes.

13. Primero escribir el HTML y después el CSS.

14. Evitar usar imágenes en el HTML más allá del contenido. Si se quiere hacer un botón con un icono de impresora es mejor crear un estilo con una imagen de fondo.

15. Utilizar colores de fondo. mejora la sensación de carga de la página y asegurara que, aunque no se carguen las imágenes, quedarán definidas las diferentes zonas de contenidos que se quieren marcar en la página.

El selector universal

Un * no sólo es un asterisco, también es el llamado selector universal y se debería utilizar como primera regla en cualquier hoja de estilo para eliminar aquellas propiedades por defecto que suelen ser perturbadoras o no sabemos de dónde vienen.

Fundamentalmente hay dos propiedades que sería bueno ponerlas a cero:

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

Colocada esa regla eliminamos *paddings* y márgenes de todas las etiquetas y por lo tanto, luego podremos asignarle los valores exactos.

Otro ejemplo donde se resetean más propiedades:

```
* {  
    border: none;  
    margin: 0;  
    outline: none;  
    padding: 0;  
    text-decoration: none;  
}
```

El selector * no sólo se utiliza de ese modo, también puede ser parte de cualquier otra regla. Por ejemplo, podríamos establecer todas las fuentes de las etiquetas dentro de un <div> de este modo:

```
div * {font-size: 20px;}
```

De ese modo, no es necesario crear reglas individuales para esos contenidos.

Otra alternativa es utilizar la propiedad **all** es una forma simplificada de resetear todas las propiedades de un elemento. Los valores posibles son:

initial cambia todas las propiedades a su valor inicial

inherit cambia todas las propiedades por las de su elemento padre

unset cambia todas las propiedades por las de su elemento padre o su valor inicial

Un ejemplo:

```
p {color: red; font-size: 24px;}  
...  
p {all: unset;}
```

Identificar enlaces por sus atributos

Usando los selectores de atributos podemos diferenciar los enlaces de modo muy preciso y, de ese modo, establecer variantes gráficas surtidas, personales, locas o casi cualquier cosa.

Por ejemplo, si quisiéramos diferenciar los enlaces internos a nuestro sitio de los enlaces externos, podríamos agregar propiedades específicas para que nuestros propios enlaces, se viesen distintos, usando parte de la URL:

```
a[href*="misitio."] {  
    /* acá las propiedades */  
}
```

Lo mismo podríamos hacer de modo inverso, establecer una regla para que lo que se vea distinto sean los enlaces externos:

```
a[href ^= "http"]:not([href *= "misitio."]) {  
    /* acá las propiedades */  
}
```

Ahora vamos a ver como identificar cierto tipo de archivo y adosarle automáticamente una imagen al vínculo. Por ejemplo, para los archivos PDF tendríamos que tener esta declaración de estilo donde utilizamos el operador \$ para seleccionar todos los vínculos cuya URL termine con .pdf

```
a[href $=".pdf"] {  
    padding-right: ancho_imagen;  
    background: transparent url(URL_imagen) no-repeat center right;  
}
```

Lo mismo, podríamos hacer para cualquier tipo de archivo o, variando el operador, seleccionar los vínculos que apunten a una dirección de correo donde ^ indica que el valor debe comenzar con el texto indicado.

```
a[href ^= "mailto:" ] {  
    padding-right: ancho_imagen;  
    background: transparent url(URL_imagen) no-repeat center right;  
}
```

Ejemplos de nth-child

Imaginemos una lista cualquiera:

```
<ul>
  <li>el primer item de la lista</li>
  <li>el segundo item de la lista</li>
  ...
  <li>el septimo item de la lista</li>
</ul>
```

Y veamos algunos ejemplo:

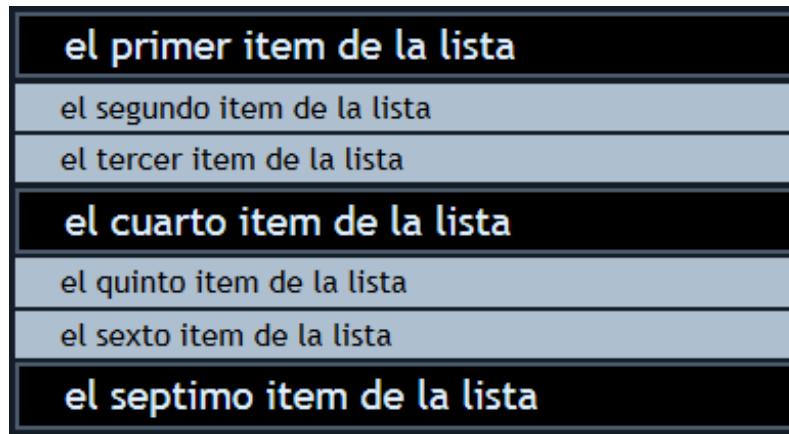
```
ul li:nth-child(3) {
  /* nth-child(3) identifica al tercer elemento */
  background-color: #6495ED;
  font-weight: bold;
  text-transform: uppercase;
}
```

el primer item de la lista
el segundo item de la lista
EL TERCER ITEM DE LA LISTA
el cuarto item de la lista
el quinto item de la lista
el sexto item de la lista
el septimo item de la lista

```
ul li:nth-child(2n+1) {
  /* nth-child(2n+1) identifica a los elementos impares [ 1 3 5 7 ] */
  background-color: #789;
  color: #FFF;
}
```

el primer item de la lista
el segundo item de la lista
el tercero item de la lista
el cuarto item de la lista
el quinto item de la lista
el sexto item de la lista
el septimo item de la lista

```
ul li:nth-child(3n+1) {  
    /* nth-child(3n+1) identifica a los elementos 1 4 y 7 */  
    background-color: #000;  
    border: 2px solid #456;  
    color: #DEF;  
    font-size: 18px;  
}
```



Eventos click y CSS

Un evento es eso que ocurre cuando hacemos algo. En CSS, el más común es el *hover*; ponemos el cursor encima de una etiqueta y, automáticamente, se ejecuta el evento.

No es algo controlable, se ejecuta siempre aunque la propiedad **pointer-events** nos permite deshabilitarlo. Por ejemplo si tuviéramos un enlace al que le adosamos esa propiedad, simplemente, no funcionaría:

```
<a href="url_pagina" style="pointer-events: none;">click acá</a>
```

Si bien *hover* es el más común no es el único; lo malo es que no existe ninguno que reaccione ante un *click*. El CSS, por alguna razón, jamás ha tenido algo semejante y ese tipo de acciones sólo pueden hacerse con JavaScript.

Sin embargo, hay algunas alternativas; ninguna de ellas es perfecta pero, por ahora, no hay muchas más. La mayoría utiliza la pseudo-clase **:target**:

```
<style>
  #democlick1 {display: none;}
  #democlick1:target {display: block;}
</style>

<a href="#democlick1">demo con target</a>

<div id="democlick1">
  ..... cualquier contenido .....
</div>
```

Otra alternativa es utilizar la pseudo-clase **:focus**; en este caso, al hacer *click*, se expandirá el contenido oculto.

```
<style>
  .democlick2 {display: none;}
  span:focus ~ .democlick2 {display: block;}
</style>

<span tabindex="0">demo con focus</span>
<div class="democlick2">
  ..... cualquier contenido .....
</div>
```

Tiene dos ventajas con respecto al anterior, por un lado, la página se queda quieta y, podemos aplicarlo a una clase lo que nos permite poner varios sin necesidad de identificarlos pero tiene dos desventajas; para volverlo a cerrar, debemos hacer *click* en cualquier parte “afuera” y además, el contenido oculto debe estar inmediatamente después, sin etiquetas intermedias.

El último método es mas sofisticado y requiere más etiquetas pero es el que funciona mejor ya que nos permite generar un efecto *toggle*, es decir, *click* y expandir y *click* contraer. En este caso, se usa la pseudo-clase :checked:

```
<style>
.democlick3 {display: none;}
:checked ~ .democlick3 {display: block;}
input.oculto[type=checkbox] {position: absolute;left: -999em;}
</style>

<label for="toggle-oculto1">demo con checked</label>
<input type="checkbox" id="toggle-oculto1" class="oculto" />
<div class="democlick3">
..... cualquier contenido .....
</div>
```