

# CSS, seriously

Diseño de Interfaces Web

Santiago González  

# TOC

— Table Of Contents —

- **Refresher**
- **La cascada!** - orden, especificidad, herencia, reset
- **CSS units** - em, rem, vw, variables
- **Box model** - border-box, margin, ::after



# Refresher

# Términos



width:700px



width:600px



# HTML semántico

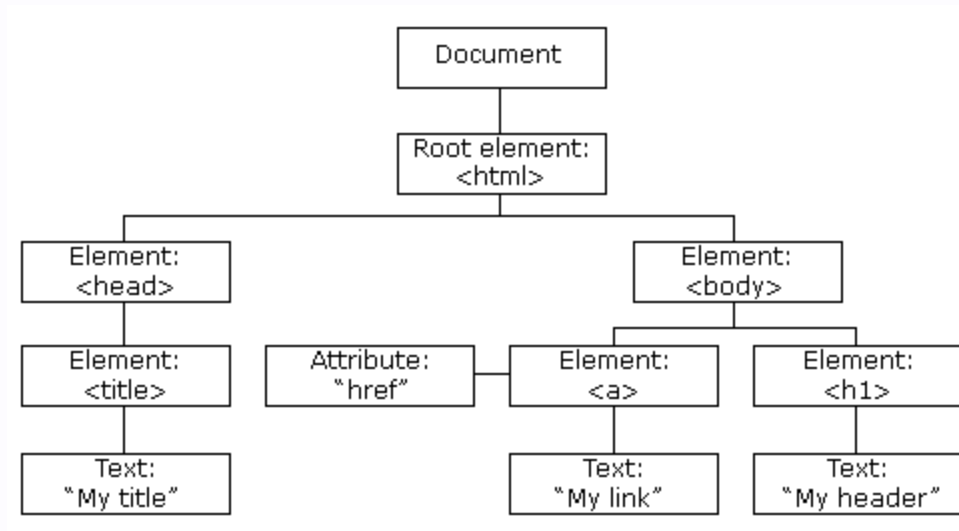
- Elementos no semánticos: `div` , `span` . No dicen nada sobre el contenido.
- Los elementos semánticos definen el tipo de contenido que incluyen. Son un contenedor igual que `div` , pero hacen automatizable el contenido (ej: lector para ciegos). Úsalos **siempre** que puedas.

Algunos comunes: `main` , `section` , `article` , `nav` , `header` , `aside` , `footer` ... **hay unos cuantos**

# DOM

**Documento Object Manager:** representación en árbol de un documento HTML.

Un **selector** CSS define el/los nodos sobre los que se aplican sus reglas.



# Selectores

Busca alguna *cheat sheet* como esta

- `a b`, `a, b`
- `a > b`, `a + b`, `a ~ b`
- `a.c`, `a .c`
- `*`, `a *`
- `a :nth-child(2) { }` vs. `a:nth-child(2) { }`
- `a:last-of-type { }`
- `a:empty { }`

Comprueba que lo controlas



# La cascada!



# Hojas de estilo

El *algoritmo de cascada* decide el valor de cada propiedad en este orden (en realidad es **más complejo**):

- **User-agent stylesheet:** el cliente (navegador) pone su **hoja de estilo por defecto**, ya sea con .css o código.
- **User stylesheet:** el navegador/plugins del navegador pueden alterar el estilo.
- **Author stylesheet:** realizadas por el diseñador web.
- **!important** : las marcadas como importantes se sitúan por encima del resto.

# Specificity

Cuando varias reglas se aplican *al mismo elemento*, la cascada decide cuál se aplica según su **especificidad**.

La especificidad es el **peso (*weight*)** de la regla, definido contando el número de ids, clases y elementos en el selector. Las pseudo clases ( `:link` ) cuentan igual.

	ids	classes	elements	weight
p	0	0	1	1
p.class	0	1	1	11
article p.class	0	1	2	12
article #id	1	0	1	101

# Specificity


```
<nav>
  <a class="elem">Element 1</a>
  <a class="elem" id="myid">Element 2</a>
</nav>
```

```
nav a.elem { color: yellow; } // weight: 012
nav .elem  { color: blue;   } // weight: 011
nav a#elem { color: pink;   } // weight: 102
nav #elem  { color: green;  } // weight: 101
a.ele     { color: red;     } // weight: 011
```

Sólo cuando dos reglas tienen el mismo peso, se decide por el **orden en el que aparecen**.

- [W3: CSS3 Cascading](#)
- [MDN: Cascading](#)

# Resumiendo...

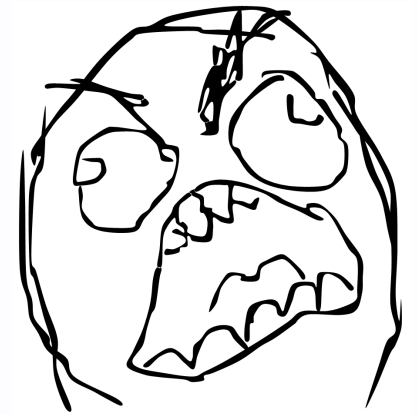
 width:800px

# Problema

¿De qué color se muestra cada párrafo?

```
#lista p { color: blue; }  
.mark { color: red; }
```

```
<div id="lista">  
  <p>Párrafo 1</p>  
  <p class="mark">Párrafo 2</p>  
</div>  
<p class="mark">Párrafo 3</p>
```



¿Cómo lo resolverías?

# Soluciones

1. `.mark { color: red !important; }`

No. No. No. No! (a continuación)

2. `#lista .mark {color: red; }`

Aumenta la especificidad del hijo (indeseable)

3. `.lista p { color: blue; }`

Baja la especificidad del padre con una nueva clase

Mejor la sol. 3: bajar la especificidad da menos problemas.

# Tip: evita **!important**

Un buen diseño NUNCA usa **!important** . Si tienes un problema en la "liga" *normal*, sólo lo trasladas a la liga *important*.

¿Dónde es "acceptable"?

- Si diseñas un tema de WordPress, no uses important.
- Pero si estás usando un tema y quieres reglas por encima sin modificar el código original, puede ser OK.



# Tip: evita IDs (debatible)

- Los IDs tienen especificidad superior a las clases. Para "vencerlas" obligas a usar `!important` o más IDs, dando lugar a `specificity wars`.
- **Reusabilidad:** Las clases se pueden reutilizar, pero los IDs deberían ser únicos.
- **Flexibilidad:** Un elemento puede tener varias clases, pero sólo un ID, por lo que es muy limitado.

¿Dónde tiene más sentido? Ej: identificar elementos en JS, pero no para CSS.

# Tip: evita el orden

El código CSS debería poderse refactorizar fácilmente. Si el resultado depende del orden se complica razonar sobre él y los efectos colaterales.

Busca la encapsulación.

Luego vemos cosas feas:

```
/* !!! NO PONER encima del bloque XXX!!! Bad shit, man. */  
.thisclass { /*...*/ }
```

Trabaja con la especificidad en vez del orden.  
(hay excepciones)

# Herencia (*inheritance*)

Si un elemento no define una propiedad, es posible que la herede de su ancestro en el DOM.

Algunas propiedades se heredan por defecto (color, font...) y otras no (padding, margin...).

- El valor `inherit` fuerza la herencia cuando no se aplica por defecto.
- Ej: el user agent pone los enlaces en azul, y nos lo podemos saltar:

```
a { color: inherit; }
```

# Herencia (*inheritance*)

- El valor `initial` hace que no se aplique la herencia, y aplica su valor por defecto (el definido en el estándar CSS, no en el user-agent).
- Usar `all: initial` hace que un elemento bloquee toda la herencia de autor y user-agent, dejando el aspecto original.

# Reseteando el estilo

El *user-agent style* es diferente entre navegadores (sobre todo en los antiguos), causando inconsistencias.

Es muy frecuente añadir en tu código algo como:

- **CSS reset:** Reescribe TODAS las reglas del user-agent (en código *author*). Hay muchas formas de reset.
- **CSS normalize:** para cada navegador, deshace todas las reglas en los que se desvía del estándar, pero el resto no las toca. Se usa mucho [normalize.css](#).

# normalize.css

Es frecuente usar un normalizador y luego un reset propio pequeño para las cosas que necesitamos.

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

Lectura: [Normalize CSS or CSS Reset?!](#)

*Tip:* te puedes descargar el .css o [enlazar un CDN](#)

Los navegadores ahora están más estandarizados

¿Sigue siendo esto necesario? 

# css units

A pixel art landscape background featuring a blue sky with white clouds, a green tree with an orange trunk, a grey mountain, and a green field with brown soil at the bottom.

Pixels, em, rem and stuff...

# Unidades absolutas

Algunas de ellas ([hay más](#))

- `px` : en CSS un pixel **no es un punto de luz**; es independiente de la resolución del monitor (retina display, HiDPI...)
- `mm` : milímetro
- `in` : pulgada
- `pt` : punto —usado en tipografía

1 in = 25.4 mm = 72 pt = 96 px



# Unidades relativas

Algunas de ellas ([hay más](#))

- `em` : tamaño fuente del elemento padre
- `rem` : tamaño fuente de la raíz del DOM
- `vw` , `vh` : 1% of viewport width/height
- `vmin` , `vmax` : 1% of viewport's smaller/larger dimension

El **computed value** es el valor absoluto resultante de aplicar uno relativo (lo calcula el navegador)

# em

Unidad relativa al tamaño del `font-size` del elemento.  
Muy útil para `padding`, `height`, `width` ... ya que se escala todo según el tamaño del texto.

```
.block {  
  font-size: 16px;  
  padding: 0.5em 1em;  
  border-radius: 1em;  
}
```

hola 16px

hola 30px

hola 60px

# em

Si se usa en el `font-size`, se toma en relación al tamaño heredado del padre.

```
<p>Hola, esto va <span class="resaltado">más grande</span></p>
```

```
.resaltado {  
  font-size: 1.5em;  
}
```

Hola, esto va **más grande**

# Cuidado con **em** !

Puede descontrolarse en anidaciones:

```
ul { font-size: 0.8em; }
```

- Primer nivel
  - Segundo nivel
    - Tercer nivel
      - Cuarto nivel

[ver el pen](#)

# rem - root em

El nodo raíz del DOM es `<html>` , que tiene la pseudo-clase `:root` . Son lo mismo, pero con diferente especificidad (tag vs. class)

`rem` es relativo al `text-size` del root e independiente de todos los padres entre medias.

```
:root {  
  text-size: 1em; // Browser default (16px)  
}  
p {  
  font-size: .8rem;  
}
```

# ¿Uso `em` o `rem` ?

Normalmente será una combinación de ambos.

En general (aprox):

- `rem` para *font-size* (no siempre!)
- `px` para bordes
- `em` para otras propiedades

# Arquitectura de componentes

```
<div class="panel large">
  <h2>Título</h2>
  <p>Lorem ipsum...</p>
</div>
```

Podemos añadir fácilmente clases como `small`.  
Esto ya parece framework CSS.  
Al usar `rem`, el panel es independiente de su padre.

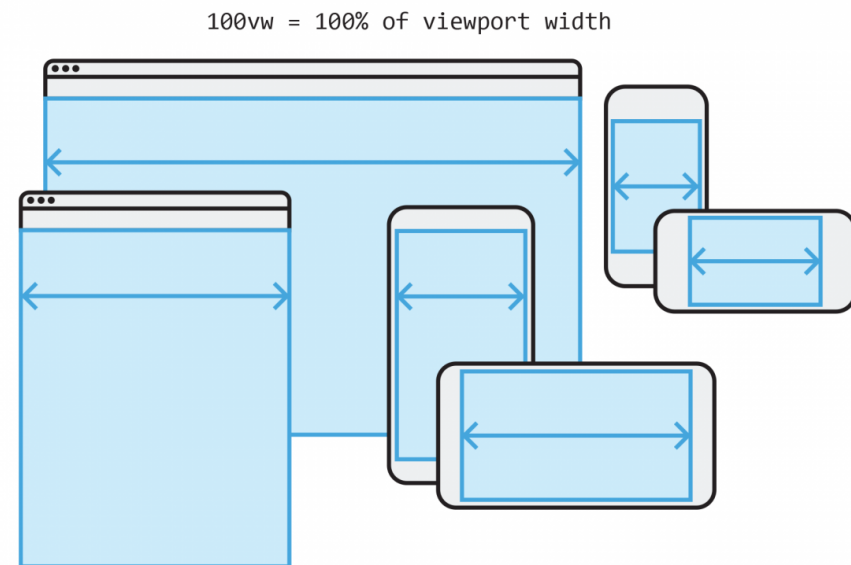
```
.panel {
  font-size: 1rem;      // La clave!
  padding: 1em;         // Depende de font-size
  border: 1px solid #999;
}
.panel > h2 {
  font-size: 1.2em;     // Depende de .panel
  font-weight: bold;
}
.panel.large {
  font-size: 1.5rem;    // Cambia la clave
}
```

# vh y vw

Hacen referencia en % al alto y ancho de la pantalla. Ej:

- Un *above the fold* que ocupe justo la pantalla.
- Un texto que se ajuste la pantalla. Ej:

```
:root {  
  // 1em hace de tamaño mínimo  
  font-size: calc(1em + 1vw);  
}
```





Ejercicio de scroll por "pantallas" en pencode.

```
background-attachment: fixed;
```

# Valores sin unidad

Algunas propiedades tienen valores numéricos sin unidad: `font-weight` (grosor de la fuente), `z-index` (orden de capas).

`line-height` controla el espaciado entre las líneas de un mismo párrafo, y acepta valores con y sin unidades.

```
section {  
  font-weight: 200; // grosor ligero  
  line-height: 1.5; // 1.5 * font-size  
}
```

# Variables CSS

Deben comenzar por dos guiones.

- Se pueden usar en elementos que estén por debajo del nodo DOM en el que se definieron.
- Si `var()` tiene dos argumentos, el segundo es el que se usa si la variable no se definió en ningún ancestro.

```
:root {  
  --primary-color: rgb(230, 25, 25); // Definición  
}  
.block {  
  color: var(--primary-color);           // Uso  
  color: var(--primary-color, red);  
}
```

# Ejemplo

A continuación vemos el código para crear estos dos paneles de forma modular.

## **Panel normal**

Lorem ipsum non pariat quod autem  
a voluptatem quasi totam voluptas.

## **Panel invertido**

Lorem ipsum non pariat quod autem  
a voluptatem quasi totam voluptas.

Puedes consultar también el [código en este pen](#)

# Ej: Panel normal

```
<div class="panel">
  <h2>Hola</h2>
  <p>Lorem ipsum...</p>
</div>
```

```
:root {
  --text-color: #333;
}
.panel {
  width: 320px;
  border: 1px solid #ccc;
}
.panel > h2,
.panel > p {
  margin: 0px;
  color: var(--text-color);
}
```

# Ej: Panel invertido

La nueva regla CSS tiene más especificidad que `.panel` para que se coja siempre su valor de `border-width` y no dependa del orden del código (no recomendable).

```
<div class="panel inverted">
  <h2>Hola</h2>
  <p>Lorem ipsum...</p>
</div>
```

```
div.inverted { /* Más específico que .panel */
  border-width: 0px;
  background-color: #111;
  --text-color: #eee; /* Redefinición! */
}
```

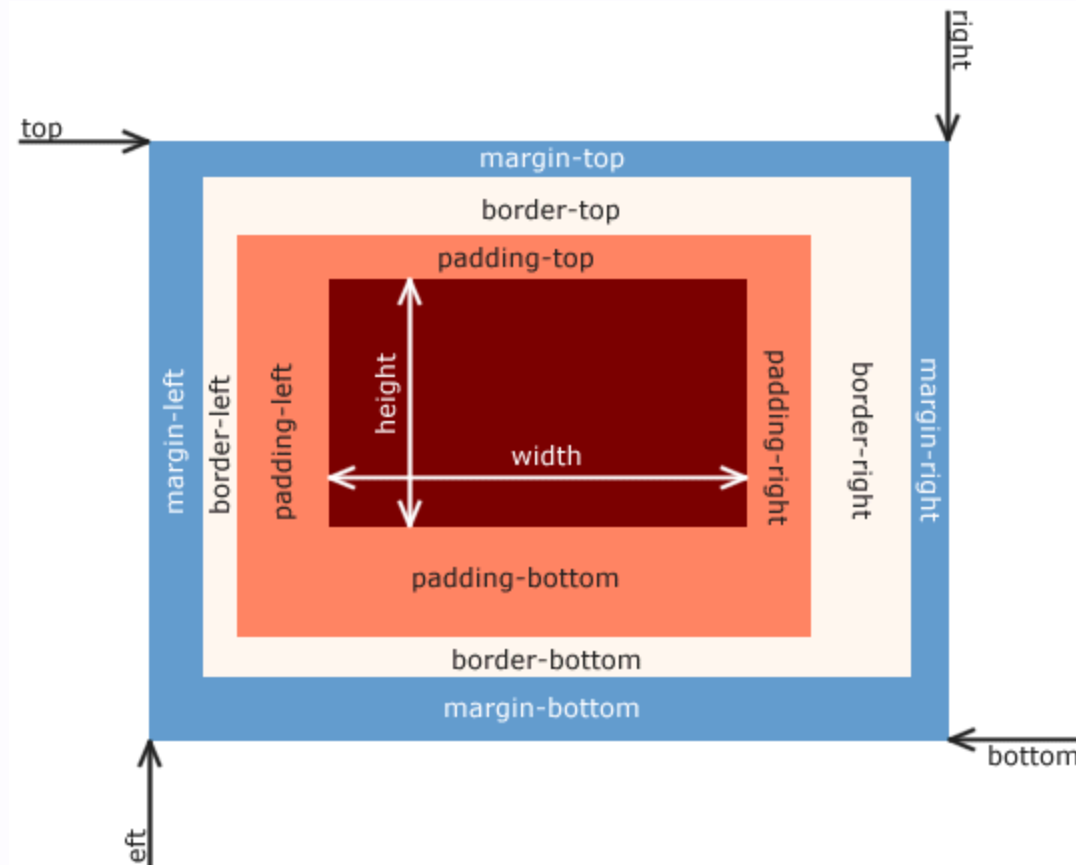
# BOX MODEL

A photograph of an open, empty cardboard box, likely a shipping container, resting on a light-colored wooden floor. The box is made of brown corrugated cardboard and is shown from a slightly elevated, angled perspective. The top flaps are open, revealing the interior. The text 'BOX MODEL' is superimposed over the center of the image in a large, bold, white, sans-serif font. The background is a dark, solid color, possibly a wall or a backdrop, which contrasts with the lighter tones of the box and floor.

# El modelo de caja CSS

CSS box: content, padding, margin, margin.

El background (imagen o color) sólo afecta al content y al padding.





# Problema!

Intenta poner dos cajas **con padding** una al lado de la otra ocupando un ancho de 70% y 30% respectivamente.

```
padding: 20px;  
width: 70%;  
float: left;
```



width: 700px

¡No cabe en una línea! ¿Soluciones con `calc` ?

# box-sizing

Hay dos modos de contabilizar el `width` de una caja:

- `box-sizing: content-box` cuenta sólo el contenido (default)
- `box-sizing: border-box` : content, border y padding

Trabajar con `border-box` resulta muy natural, aunque no sea el comportamiento por defecto.

Con border-box, el problema de las cajas al 70-30% de anchura con padding resulta trivial, ya que el width ya incluye el padding.  
ver [este pen](#).

# border-box everywhere

Es frecuente encontrar:

```
/* Selector *: todos los elementos */  
* {  
    box-sizing: border-box;  
}
```

Mejor aún, esto nos permite "cortar" el uso de border-box en una sub-rama del DOM (imagen siguiente):

```
:root {  
    box-sizing: border-box;  
}  
* {  
    box-sizing: inherit;  
}
```



# Margin collapsing

Los márgenes verticales (nunca horizontales) de dos cajas juntas se solapan en algunas ocasiones.

- Dos boxes hermanas solapan sus márgenes.
- Un padre y un hijo solapan márgenes si el padre no tiene padding ni margin.
- Una caja vacía con sólo márgenes se solapa por completo con una hermana.

# Margin collapsing

Hermanos, padre y hermano vacío.



width:1000px

Si el padre tiene un borde o padding, no se solapa nada.

Tutorial y codepen: [nanajeon.com/cutup-2-margin](https://nanajeon.com/cutup-2-margin)

# Margen negativo

**Contact card:** juega con el margen negativo para que este código HTML quede como se muestra.

```
<div class="card">
  <div class="card-top"></div>
  
  <div class="card-body">
    <h2>Lorem Ipsum</h2>
    <p>Non eligendi...<p>
  </div>
</div>
```





# Propiedad `display`

- `display: inline` insertado en el texto como `span`. Sus dimensiones son por el contenido, y `width` y `height` no le afectan.
- `display: block` es como `div`. Comienza siempre en una nueva línea y su tamaño es modificable.
- `display: inline-block` es *inline* pero se le afectan `width` y `height`.
- `display: none` como si no estuviera en el DOM.

Muchos más en [w3schools > display property](#)

# ::before y ::after

Son pseudo-elementos (no pseudo-clases, ojo a la especificidad) que se añaden como primer/último hijo.

```
a::after { content: "→" }
```

Es `inline` por defecto.

Ver código en este [pen](#) 

# Contact card, revisited

El contenedor `card-top` que hicimos no aporta NADA de contenido, por lo que no debería estar en HTML.

```
<div class="card">
  <div class="card-top"></div> <!-- quitar! -->
  <!-- ... -->
</div>
```

Modifica tu código para usar `::before` ([solución](#)):

```
.card::before {
  content: "";
  display: block;
  /* ... */
}
```

# Truqui

Puedes añadir un atributo personalizado con texto secundario que se vaya a incluir en *after/before*.

```
<div somedata="Info extra"><!-- ... --></div>
```

```
div::after {  
  content: attr(somedata);  
}
```

No debería ser contenido importante, ya que no es accesible y cosas como lectores de HTML no lo entienden.