CSS AVANZADO

1. Unidades de longitud: absoluta y relativa

Link a la especificación del W3C.

Las **unidades de longitud absoluta** no son relativas a nada más y, en general, se considera que siempre tienen el mismo tamaño. Son las siguientes:

Unidad	Nombre	Equivale a
cm	Centímetros	1cm = 96px/2,54
mm	Milímetros	1mm = 1/10 de 1cm
Q	Cuartos de milímetros	1Q = 1/40 de 1cm
in	Pulgadas	1in = 2,54cm = 96px
pc	Picas	1pc = 1/6 de 1in
pt	Puntos	1pt = 1/72 de 1in
px	Píxeles	1px = 1/96 de 1in

La mayoría de estos valores son más útiles cuando se usan en una salida en formato impreso que en la salida de pantalla. Por ejemplo, normalmente no usamos *cm* (centímetros) en pantalla. El único valor que quizá uses frecuentemente es *px* (píxeles).

Las **unidades de longitud relativa**, en cambio, son relativas a otra cantidad. La ventaja de usar unidades relativas es que con una planificación cuidadosa puedes lograr que el tamaño del texto u otros elementos escalen en relación con todo lo demás en la página. En la tabla siguiente se enumeran algunas de las unidades más frecuentes:

Unidad	Relativa a
%	Tamaño de su elemento padre.
em	Tamaño de letra del elemento padre (en el caso de propiedades tipográficas como `font-size`) y tamaño de la fuente del propio elemento en el caso de otras propiedades, como `width`.
rem	Tamaño de la letra del elemento raíz (`<html>`). Abreviatura de *"root em"*.
vw	1% de la anchura del viewport (`vh` es *viewport width*).
vh	1% de la altura del viewport (`vh` es *viewport height*).
vmin	1% de la dimensión más pequeña del viewport. Es igual al mínimo entre `vw` y `vh`.
vmax	1% de la dimensión más grande del viewport. Es igual al máximo entre `vw` y `vh`. Observa el siguiente ejemplo de `vmin` y `vmax` en codepen.
ex	Altura de la letra 'x' de la fuente del elemento. Útil cuando se desea cambiar el tamaño de algo en relación a la altura de las letras minúsculas del texto.
ch	La anchura del carácter '0' (*cero*) en el tipo de fuente utilizado. Útil cuando se muestra texto, dado que no se recomienda pasar de 75 caracteres por línea para mejor legibilidad. Por ejemplo: `max-width: 60ch` significa que se mostrarán 60 caracteres por línea como máximo.

1.1. Diferencias entre `em` y `rem`

Sin duda `em` y `rem` son las dos longitudes relativas que encontrarás con mayor frecuencia. Se utilizan tanto para tamaños de letra (especialmente `rem`) como para establecer `margin` y `padding`. Hay que entender cómo funcionan y cuáles son las diferencias entre ellas.

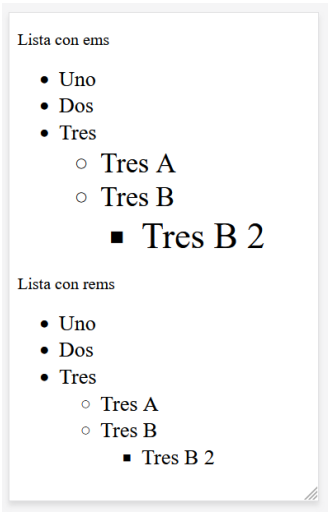
1.1.1. Diferencias al usarlas como tamaño de fuente

Observa el siguiente ejemplo, también disponible en codepen.

```
<style>
  html     { font-size: 16px;   }
  .ems li  { font-size: 1.5em;  }
  .rems li { font-size: 1.5rem; }
</style>

<p>Lista con ems</p>
<ul class="ems">
  <li>Uno</li>
  <li>Dos</li>
  <li>Tres
    <ul>
      <li>Tres A</li>
      <li>Tres B
        <ul>
          <li>Tres B 2</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>

<p>Lista con rems</p>
<ul class="rems">
  <li>Uno</li>
  <li>Dos</li>
  <li>Tres
    <ul>
      <li>Tres A</li>
      <li>Tres B
        <ul>
          <li>Tres B 2</li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```



El HTML está compuesto por dos listas anidadas exactamente iguales, salvo que la primera tiene la clase `ems` y la segunda la clase `rems`.

Fíjate en los estilos. Para empezar, configuramos un tamaño de letra de 16px en el elemento raíz `<html>`.

- **La unidad `em` se calcula en relación a su elemento padre.** Los elementos `<li>` dentro de un elemento `<ul>` toman el tamaño con respecto a su elemento padre. Por lo tanto, el tamaño de letra aumenta progresivamente en cada nivel de anidamiento, ya que en cada uno el tamaño de letra está establecido en 1.5em (1,5 veces el tamaño de letra de su elemento padre). Es por esta razón por la que yo no recomendaría usar `em` para tamaños de fuente, salvo que estemos seguros de que este problema de anidamiento no va a ocurrir.

- **La unidad `rem` se calcula en relación al del elemento raíz** (`rem` es la abreviatura de *root em*). Los elementos `<li>` dentro de un elemento `<ul>` toman su tamaño del elemento raíz (`<html>`). Por tanto, el tamaño de letra no aumenta en cada nivel de anidamiento. Si no se ha establecido un tamaño de fuente para el elemento raíz el navegador utilizará un tamaño de fuente predeterminado, que suele ser 16px. Esto lo puedes comprobar en el ejemplo anterior eliminando el estilo dado a `html`.

Si cambias el atributo `font-size` de `<html>` en el CSS verás que todo lo demás cambia en relación con él, tanto la letra cuyo tamaño está especificado en unidades `rem` como la que lo está en unidades `em`.

1.1.2. Diferencias al usarlas como `margin` o `padding`.

Hay una pequeña diferencia cuando se usa `em` en `margin` o `padding`. En vez de tomar como referencia el valor de `font-size` de su padre, lo toma de sí mismo.

Observa este ejemplo, también disponible en codepen:

```
<style>
  .btn {
    background-color: black;
    color: white;
    padding: 2em;
  }

  .btn-xl {
    font-size: 2em;
  }

  .btn-l {
    font-size: 1em;
  }

  .btn-s {
    font-size: 0.5em;
  }

  code {
    background-color: lightgray;
    padding: 2px 4px;
  }
</style>

<button class="btn btn-xl">Botón</button>
<button class="btn btn-l">Botón</button>
<button class="btn btn-s">Botón</button>
```

Observa la diferencia de poner `padding: 2em` a `padding: 2rem` en la clase `btn`.

Fijando el `padding` en `ems` logramos que se adapte al tamaño de fuente del propio botón, por lo que puedo definir botones de varios tamaños y todos los `margins` y `paddings` serán proporcionados.

Dado que, como hemos dicho, los navegadores fijan su tamaño de letra en 16px, puede resultar un poco confuso usar `em` o `rem`, ya que tenemos que hacer cálculos mentales para pasar esas unidades a píxeles. Un **pequeño truco** que nos facilita la vida en este aspecto es fijar como tamaño de letra de `html` el 62.5%:

```
html { font-size: 62.5%; }
```

Con esto conseguimos fijar nuestro `root` en 10px, por lo que ahora 15px es simplemente 1.5rem, 20px es 2rem, etc.

1.2. Nuevas unidades en 2022

A partir de marzo de 2022 se incorporaron nuevas unidades como parte del esfuerzo conjunto de los desarrolladores de navegadores llamado *Interop 2022*.

Se pretende que estas nuevas unidades CSS reflejen mejor las dimensiones mínimas, máximas y actuales de la ventana gráfica. Para los desarrolladores, esto solía requerir tanto CSS como JavaScript.

Estas son las unidades descritas:

- svh
- lvh
- dvh
- svw
- lvw
- dvw
- svmin
- lvmin
- dvmin
- svmax
- lvmax
- dvmax
- vi
- svi
- lvi
- dvi
- vb
- svb
- lvb
- dvb

Tienes la descripción de cada una de ellas en el siguiente enlace:

https://www.terluinwebdesign.nl/en/css/incoming-20-new-css-viewport-units-svh-lvh-dvh-svw-lvw-dvw/

1.3. Webgrafía

- https://lenguajecss.com/css/modelo-de-cajas/unidades-css/
- https://www.w3schools.com/cssref/css_units.php
- https://developer.mozilla.org/es/docs/Learn/CSS/Building_blocks/Values_and_units
- https://www.w3.org/TR/css-values-3/#lengths
- https://www.terluinwebdesign.nl/en/css/incoming-20-new-css-viewport-units-svh-lvh-dvh-svw-lvw-dvw/

2. Especificidad y reglas de prevalencia

Cuando tenemos varias reglas CSS iguales que afectan a un mismo elemento, ¿cómo se decide qué regla se aplica? Observa el siguiente ejemplo:

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

<p style="color: red">Hola mundo</p>
```

¿De qué color será el texto del párrafo? La respuesta es rojo y el motivo es que el CSS en línea (*inline*) prevalece sobre el CSS interno (`<style>`) y éste sobre los externos.

¿Y si aparecen en el mismo sitio? Por ejemplo:

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

En este caso se aplicará el último estilo que el navegador encuentre, por lo que en este caso pintará el párrafo de rojo.

Ahora bien, imagina el siguiente escenario:

```
<style>
  p.parrafo {color: red;}
  p         {color: blue;}
  .parrafo  {color: greenn !important;}
</style>

<p class="parrafo">Hola mundo</p>
```

Aquí estamos usando 3 selectores distintos que encajan en nuestro párrafo, ¿cuál de ellos se aplica? El sentido común nos dice que debería aplicarse el primero. Mientras que `p` selecciona todos los párrafos y `.parrafo` todos los elementos con clase párrafo, `p.parrafo` selecciona los párrafos que tienen como clase párrafo, siendo el más específico de todos. Pero claro, los navegadores no pueden aplicar el sentido común, así que aplican las matemáticas. Aquí es donde entra en juego el concepto de **especificidad**.

La especificidad es la **forma en la que los navegadores deciden qué declaración CSS debe ser aplicada cuando varias hacen referencia al mismo elemento**.

La especificidad se representa mediante un valor (o *peso*) que se le asigna a una declaración CSS y se calcula en base a **tres valores**: (X, Y, Z):

- X: Número de veces que aparece un `#id` en el selector.
- Y: Número de veces que aparece una `.clase`, `:pseudoclase` o `[atributo]` en el selector.
- Z: Número de veces que aparece un `tag` o un `::pseudoelemento` en el selector.

Para saber si una declaración CSS es más específica que otra (y, por lo tanto, será aplicada) sólo hay que calcular los valores anteriores. Entre los tres valores forman un número de 3 cifras que indica la especificidad. **El número más alto es más específico** y será el aplicado. En caso de empate, el último que aparezca en el código será aplicado.

Volviendo al ejemplo anterior, podemos calcular fácilmente la especificidad de cada uno de los selectores:

```css
/* Especificidad (0,1,1) */
p.parrafo {color: red;}

/* Especificidad (0,0,1) */
P {color: blue;}

/* Especificidad (0,1,0) */
.parrafo {color: green;}
```

Para declaraciones más complejas o para practicar puedes utilizar la siguiente calculadora web de especificidad: https://specificity.keegan.st/

Y una ilustración, descargable en formato imagen o pdf: https://specifishity.com/

Sin embargo, existe una instrucción que altera todo lo que hemos dicho hasta ahora. Es **la instrucción** `!important`. Esta instrucción, situada al final de una declaración, siempre tiene prevalencia sobre cualquier otra. No obstante, solo deberías usarla en declaraciones específicas de CSS que sobrescriban CSS de librerías externas, como Bootstrap.

En resumen, para resolver conflictos en CSS se sigue este orden: primero se mira si alguna regla tiene `!important`, luego se considera la especificidad y, finalmente, se mira el orden de declaración. Comprender la especificidad y las reglas de prevalencia te permitirá controlar de manera efectiva el estilo de tus elementos HTML y solucionar conflictos de manera sistemática.

2.1. Webgrafía

- W3C: https://www.w3.org/TR/selectors-3/#specificity
- https://developer.mozilla.org/es/docs/Learn/CSS/Building_blocks/Cascade_and_inheritance#especificidad
- https://lenguajecss.com/css/cascada-css/que-es-cascada/
- Calculadora de especificidad: https://specificity.keegan.st/
- https://specifishity.com/

3. Funciones

En CSS a veces necesitaremos un sistema de apoyo para realizar operaciones más cercanas a un lenguaje de programación que a un lenguaje de estilos, como hacer cálculos o utilizar valores previamente calculados. Por esta razón, con el tiempo, se han ido añadiendo mejoras, como las *custom properties*, un mecanismo similar a unas variables CSS.

En esta sección vamos a ver algunas funciones CSS, no todas las que existen, que no son más que funciones de apoyo que podemos utilizar en CSS para realizar cálculos u operaciones sencillas de una forma fácil y sencilla, sin tener que abandonar CSS.

Un resumen de las funciones CSS que veremos:

- `calc()`: Permite realizar operaciones aritméticas con unidades CSS como `px`, `%`, `vw`, `vh` u otras (incluso combinadas).
- `min()`: Permite calcular el valor mínimo de las unidades indicadas.
- `max()`: Permite calcular el valor máximo de las unidades indicadas.
- `clamp()`: Permite calcular valores "ajustados". Equivalente a `max(MIN, min(VAL, MAX)`).
- `attr()`: Permite obtener el valor de un atributo desde CSS.

3.1. `calc()`

Es posible que en algunas ocasiones necesitemos indicar valores precalculados por el navegador. Por ejemplo, la suma de dos valores que a priori desconocemos o no sabemos exactamente cuánto suman, pero que el navegador si debería conocer. Esto es posible hacerlo con la `función calc()` de CSS, como se muestra a continuación:

```
.elemento {
  width: calc(200px + 1em);
}
```

Se pueden usar operaciones como sumas, restas, multiplicaciones o divisiones que utilicen alguna de las unidades soportadas por CSS, como números, dimensiones, porcentajes, tiempos, ángulos, etc.

Veamos un ejemplo práctico (disponible en codepen). Digamos que queremos que la fuente de nuestros párrafos sea grande en una pantalla de escritorio, pero pequeña en una pantalla móvil. Por supuesto, podemos hacerlo con *media queries*, pero con `calc()` es más rápido.

```
p {
  font-size: calc(10px + 2vw);
}
```

Prueba a redimensionar la pantalla del navegador para ver cómo varía el tamaño de letra de los párrafos.

3.2. `min()` y `max()`

Las funciones `min()` y `max()` permite elegir el valor más pequeño/grando entre de dos o más unidades pasadas por parámetro.

Observa el ejemplo siguiente, disponible en codepen, sobre `min()` y `max()`.

```
<div class="uno">Hola Mundo</div>
<div class="dos">Hola Mundo</div>

<style>
  div {
    background-color: beige;
    padding: 20px;
    font-size: 2rem;
    border: 1px solid black;
    margin-bottom: 10px;
  }

  .uno {
    width: min(200px, 40%);
  }

  .dos {
    width: max(200px, 40%);
  }
</style>
```

Se aplicará al primer `div` un `width` de 200px siempre y cuando el 40% del elemento padre sea más de 200px. Para el segundo sería al contrario, aplicará un `width` del 40% del elemento padre siempre y cuando sea menos de 200px.

Este tipo de cosas antes había que realizarlo con JavaScript.

3.3. `clamp()`

Esta función permite definir un rango de valores para una propiedad CSS, como el tamaño de fuente, el ancho de un contenedor o cualquier otra propiedad numérica. Su estructura es:

```
clamp(min, preferido, max);
```

- **min:** Es el valor mínimo que se utilizará para la propiedad.
- **preferido:** Es el valor preferido que deseas que se aplique a la propiedad. En condiciones normales se utilizará este valor.
- **max:** Es el valor máximo que se utilizará si las condiciones hacen que el valor sea más grande que el preferido. Este valor actúa como un límite superior.

Esta función es útil para crear diseños web adaptativos, dado que permite especificar un tamaño preferido mientras establece límites mínimo y máximo. Esto garantiza que el diseño se ajuste de manera adecuada a diferentes tamaños de pantalla sin que el contenido sea demasiado pequeño o demasiado grande.

Por ejemplo:

```
font-size: clamp(16px, 5vw, 24px);
```

Aquí:

- 16px es el tamaño de fuente mínimo.
- 5vw es el tamaño preferido, que escala según el ancho de la ventana.
- 24px es el tamaño de fuente máximo.

El tamaño de fuente será de `5vw` (el valor preferido) si este valor es mayor de `16px` y menor de `24px`. Si alcanza los límites, dejará de crecer/decrecer.

Esta función se podría escribir de otra manera:

```
max(MIN, min(VAL, MAX));
```

Por ejemplo, estas dos instrucciones son equivalentes.

```
width: max(400px, min(75%, 700px));
width: clamp(400px, 75%, 700px);
```

Lo que significa esta expresión es: “El valor de `width` será del 75% de su elemento padre, pero nunca será menor de 300px ni mayor de 500px”.

De esta forma, el navegador realiza lo siguiente:

- Obtiene el valor mínimo entre el segundo y tercer parámetro.
- Obtiene el valor máximo entre el primer parámetro y el resultado anterior.
- Utiliza el resultado de la operación anterior en el `width`.

Con un ejemplo se verá mejor. El siguiente código está disponible en codepen.

```
<div class="box">clamp(400px, 75%, 700px);</div>
<div class="msg"></div>

<style>
  div.box {
    width: clamp(400px, 75%, 700px);
    background-color: coral;
    padding: 50px 0;
  }
</style>
```

```
<script>
  window.addEventListener("resize", function () {
    var boxWidth = document.querySelector(".box").offsetWidth;
    var bodyWidth = document.body.offsetWidth;
    var porcentaje = (boxWidth / bodyWidth) * 100;
    var mensaje =
      "<p>box_width = " + boxWidth + "</p>" +
      "<p>body_width = " + bodyWidth + "</p>" +
      "<p>porcentaje = " + porcentaje.toFixed(1) + "%</p>";
    document.querySelector(".msg").innerHTML = mensaje;
  });
</script>
```

Modificando la anchura de la ventana observamos que la anchura del `div` es el 75% del valor del viewport siempre que este valor esté entre 400px y 700px.

3.4. `attr()`

La función `attr()` permite obtener el valor de un atributo HTML y utilizarlo desde CSS, generalmente para temas de decoración. Esta función CSS es antigua y la mejor soportada por los navegadores.

Por ejemplo, observa el siguiente fragmento de código, disponible en codepen:

```
<div class="element" data-autor="Juan">Lorem ipsum dolor sit amet</div>

<style>
  .element::before {
    content: attr(data-autor) ": ";
    color: #777;
  }
</style>
```

Por medio de un pseudoelemento CSS y utilizando la propiedad `content`, le pedimos al navegador que utilice el metadato del atributo `data-autor` del elemento donde se le está aplicando el CSS para mostrarlo como contenido.

Los atributos `data-*` son atributos personalizados (*custom attributes*) introducidos en HTML5 para que los desarrolladores pudieran alojar datos propios en etiquetas HTML. Tienes más información en:

https://www.w3schools.com/tags/att_global_data.asp

4. Selectores

Nunca está de más tener la especificación a mano.

A estas alturas ya deberías conocer los selectores CSS básicos. No obstante, vale la pena recordarlos:

Selector	Ejemplo	Selecciona
`*`	`*`	Selector universal. Selecciona todos los elementos.
`#id`	`#container`	El elemento con `id="container"`. Debería ser único.
`.class`	`.intro`	Todos los elementos con clase `intro`.
`tag`	`p`	Todos los elementos `<p>`.
`tag#id`	`p#container`	El elemento `<p>` con `id="container"`. Debería ser único.
`tag.class`	`p.intro`	Todos los elementos `<p>` con clase `"intro"`.

4.1. Selectores de atributos

Link a la especificación.

En algunos casos puede que interese dar estilos por atributos. En el ejemplo siguiente existe un atributo *disabled*. Con los atributos CSS podríamos seleccionar los elementos que contienen un atributo específico.

```
<button>Botón activo</button>
<button disabled>Botón desactivado</button>

<style>
  /* Selecciona todos los botones */
  button {}

  /* Selecciona botones con atributo disabled */
  button[disabled] {}

  /* Cualquier elemento con atributo disabled */
  [disabled] {}
</style>
```

Puedes ver una lista de selectores de por atributos en la siguiente tabla:

Selector	Representa	codepen
`[att]`	Representa un elemento con el atributo `att`, cualquiera que sea el valor del atributo.	-
`[att=val]`	Representa un elemento con el atributo `att` cuyo valor es exactamente `"val"`.	-
`[att~=val]`	Representa un elemento con el atributo `att` cuyo valor es una lista de palabras separadas por espacios en blanco, siendo una de las cuales exactamente `"val"`.	link
`[att|=val]`	Representa un elemento con el atributo `att`, cuyo valor es exactamente `"val"` o comienza con `"val"` seguido inmediatamente de `"-"` (U 002D). Esto tiene como objetivo principal permitir coincidencias de subcódigos de idioma (es-ES, es-MX, etc.).	link
`[att^=val]`	Representa un elemento con el atributo `att` cuyo valor comienza con el prefijo `"val"`.	link
`[att$=val]`	Representa un elemento con el atributo `att` cuyo valor termina en `"val"`.	link
`[att*=val]`	Representa un elemento con el atributo `att` cuyo valor contiene al menos una instancia de la subcadena `"val"`. Si `"val"` es la cadena vacía, entonces el selector no representa nada.	link

Todos estos selectores son sensibles a mayúsculas y minúsculas (***case sensitive***). Por ejemplo, el siguiente selector no incluiría archivos terminados en .PDF.

```
[href$=".pdf"]
```

Para indicar que no se distingan entre mayúsculas y minúsculas debemos añadir una 'i' antes del cierre ']' del atributo:

```
a[href$=".pdf" i]
```

Veamos algunos ejercicios. Selecciona todos los elementos…

1. Con un atributo `"href"` que contiene la palabra `"pdf"`.
2. Con un atributo `"data-category"` que comienza con `"featured"`.
3. Con un atributo `"class"` que termina con `"active"`.
4. Con un atributo `"role"` que es exactamente igual a `"main-header"`.
5. Con un atributo `"title"` que contiene la palabra `"image"`.
6. Con un atributo `"role"` que comienza con la palabra `"button"`.
7. Con un atributo `"data-id"` que comienza con `"item-"`.
8. Con un atributo `"type"` que es exactamente igual a `"submit"`.
9. Con un atributo `"aria-label"` que termina con `"close"`.
10. Con un atributo `"src"` que contiene la extensión `".jpg"`.

Soluciones:

1. `[href*="pdf"]`
2. `[data-category^="featured"]`
3. `[class$="active"]`
4. `[role="main-header"]`
5. `[title~="image"]`
6. `[role^="button"]`
7. `[data-id^="item-"]`
8. `[type="submit"]`
9. `[aria-label$="close"]`
10. `[src*=".jpg"]`

4.2. Combinadores básicos: *espacio*, >, +, ~, *

Un combinador CSS es un símbolo que permite combinar dos o más selectores formando uno más complejo y potente. Existen varios combinadores en CSS:

Nombre	Símbolo	Ejemplo	Selecciona
Combinador descendiente	(espacio)	`#page div`	Descendientes (cualquier nivel). Todos los `div` que sean descendientes de `#page`.
Combinador hijo	>	`#page > div`	Hijos directos (primer nivel). Todos los `div` que sean hijos directos de `#page`.
Combinador hermano adyacente	+	`#page + div`	Hermanos adyacentes. Selecciona todos los `div` que siguen a `#page` (`div` y `#page` son hermanos y `div` aparece justo después que `#page`).
Combinador hermano general	~	`#page ~ div`	Hermanos. Igual que el anterior, pero `div` solo debe aparecer después, no necesariamente justo después. Nota: No existe un selector CSS para hermanos previos.
Combinador universal	*	`#page *`	Selecciona todos los elementos (cualquier nivel).

Para explicar mejor la diferencia entre el combinador '+' y el '~', considera el siguiente ejemplo, disponible en codepen:

```
<p>Soy el párrafo 1</p>
<p>Soy el párrafo 2</p>
<h3>Soy un h3</h3>
<p>Soy el párrafo 3</p>
<p>Soy el párrafo 4</p>
```

El siguiente selector cambia el color de fondo sólo del párrafo 3:

```
h3 + p { background-color: yellow; }
```

Mientras que el siguiente selector cambia el color del texto del párrafo 3 y del 4:

```
h3 ~ p { color: red; }
```

4.3. Combinadores lógicos

Link a la especificación.

CSS proporciona una serie de mecanismos para agrupar o combinar selectores de una forma potente y flexible, los combinadores lógicos.

Los combinadores lógicos permiten seleccionar elementos con ciertas restricciones. Técnicamente son pseudoclases (que veremos un poco más adelante) a las que se le pueden pasar parámetros.

Puedes encontrar más información y ejemplos en:

- https://developer.mozilla.org/en-US/docs/Web/CSS/:is
- https://developer.mozilla.org/en-US/docs/Web/CSS/:where
- https://developer.mozilla.org/en-US/docs/Web/CSS/:has
- https://developer.mozilla.org/en-US/docs/Web/CSS/:not

4.3.1. `:is()`

La pseudoclase `:is()` es un reemplazo práctico de la agrupación de selectores mediante comas. Observa el siguiente ejemplo, disponible en codepen:

```
<div class="container">
  <div class="uno">Uno</div>
  <div class="dos">Dos</div>
  <div class="tres">Tres</div>
</div>
<div class="uno">Uno</div>
<div class="dos">Dos</div>
<div class="tres">Tres</div>
```

Si queremos poner de color rojo los elementos con clase `uno` y con clase `dos`:

```
.container .uno,
.container .dos {
  color: red
}
```

Podemos abreviarlo así:

```
.container :is(.uno, .dos) {
  color: red
}
```

Es importante el espacio entre `.container` y `:is()`. Con el espacio seleccionamos todos los elementos con clase `.uno` y `.dos` que sean descendientes de un elemento con clase `.container`. Sin el espacio, seleccionamos todos los elementos con clase `.uno` y `.dos` que además tengan clase `.container`.

Antiguamente, esta pseudoclase era conocida como `:matches()`, pero finalmente fue renombrada a `:is()`, por lo que es posible que nos la encontremos de esta forma si accedemos a documentación antigua.

4.3.2. `:where()`

El combinador `:where()` funciona exactamente igual que el combinador `:is()`, la única diferencia que tiene es la especificidad. Mientras que la especificidad de `:is()` es el valor más alto de su lista de parámetros, en el caso de `:where()` es siempre cero.

4.3.3. `:has()`

El combinador `:has()` permite seleccionar elementos padres basándonos en una selección de elementos hijos. Veamos algunos ejemplos:

Selector	Descripción
`a:has(> img)`	Selecciona elementos `<a>` que tengan un hijo `<img>`.

`a:has(+ img)`
 	
`a + img`
	Selecciona elementos `<a>` que tengan un hermano adyacente `<img>`.

`section:not(:has(h1,h2))`	Selecciona elementos `<section>` que no contienen ningún elemento `<h1>` o `<h2>`.
`section:has(:not(h1,h2))`	Selecciona elementos `<section>` que contienen al menos un elemento que no sea ni `<h1>` ni `<h2>` como hijos.

Algunos detalles interesantes sobre `:has()`:

- La pseudoclase `:has()` no se puede anidar dentro de otra `:has()`.
- Los pseudoelementos `::before` o `::after` no funcionan dentro de `:has()`.
- La especificidad de `:has()` es el valor más alto de los selectores indicados por parámetro.

4.3.4. `:not()`

La pseudoclase `:not()` permite seleccionar elementos que no cumplan los criterios indicados en sus parámetros.

Por ejemplo, la siguiente declaración selecciona todos los párrafos que no tengan como clase `.rojo`.

```
p:not(.rojo) { }
```

Se puede indicar una lista de criterios por parámetro, no uno solo como en el ejemplo anterior. Observa el siguiente ejemplo, también disponible en codepen:

```
<p class="red big">red big</p>
<p class="red small">red small</p>
<p class="green medium">green medium</p>
```

```
<style>
  p:not(.green, .small) { color: red; }
</style>
```

En este ejemplo sólo se verá rojo el primer párrafo, dado que no tiene ni la clase `green` ni la clase `small`.

Algunos detalles adicionales sobre la pseudoclase `:not()`:

- Los pseudoelementos como `::before` o `::after` no se pueden pasar por parámetro.
- Al igual que con `:is()`, la especificidad de `:not()` es el valor más alto de sus parámetros.

4.4. Pseudoclases

Las pseudoclases se utilizan para hacer referencia a elementos HTML que tengan un comportamiento concreto. Así, podremos seleccionar elementos que parecen iguales, pero tienen diferentes características de comportamiento. Hay muchísimas pseudoclases, las más importantes son:

Pseudoclase	Descripción
`:hover`	Selecciona el elemento si el ratón sobre dicho elemento.
`:active`	Selecciona el elemento si el usuario se encuentra pulsándolo.
`:focus`	Selecciona el elemento cuando tiene el foco (está en primer plano).
`:link`	Selecciona un elemento que es un enlace no visitado aún.
`:visited`	Selecciona un elemento que es un enlace ya visitado.
`:root`	Aplica estilo al elemento raíz (padre) del documento HTML.
`:first-child`	Primer elemento hijo (de cualquier tipo).
`:last-child`	Último elemento hijo (de cualquier tipo).
`:nth-child(n)`	Elemento hijo número *n* (de cualquier tipo).
`:nth-last-child(n)`	Elemento hijo número *n* empezando por el final (de cualquier tipo).
`:first-of-type`	Primer elemento hijo (de su mismo tipo).
`:last-of-type`	Último elemento hijo (de su mismo tipo).
`:nth-of-type(n)`	Elemento hijo número *n* (de su mismo tipo).
`:nth-last-of-type(n)`	Elemento hijo número *n* empezando desde el final (de su mismo tipo).
`:only-child`	Elemento que es hijo único (de cualquier tipo).
`:only-of-type`	Elemento que es hijo único (de su mismo tipo).
`:empty`	Elemento vacío (sin hijos, ni texto).
`:checked`	Selecciona el elemento cuando el campo está seleccionado.
`:enabled`	Selecciona cuando el campo del formulario está activado.
`:disabled`	Selecciona cuando el campo del formulario está desactivado.

La pseudoclase `:root` se refiere al elemento raíz del documento HTML, la etiqueta `<html>`. Cuando se utiliza `:root` en lugar de `<html>` es porque, al ser una pseudoclase, tiene una especificidad CSS más alta (0,1,0) que `<html>` (0,0,1), por lo que en caso de sobreescritura `:root` tendría preferencia. Además, `:root` suele usarse para establecer variables CSS (las veremos más adelante) que afectan a todo el documento.

Otras pseudoclases, quizá menos utilizadas, son:

Pseudoclase	Descripción
`:focus-within`	Selecciona el elemento si uno de sus miembros hijos ha ganado el foco.
`:focus-visible`	Selecciona el elemento cuando tiene el foco sólo de forma visible (TAB, por ejemplo).
`:target`	Selecciona un elemento que coincide con el ancla de la URL actual. Si en nuestra URL tenemos el ancla `#section1`, entonces se seleccionará el elemento con `id="section1"`.
`:lang(es)`	Selecciona elementos con el idioma español (atributo `lang="es"`)
`:dir(value)`	Selecciona elementos con la dirección indicada (`ltr` o `rtl`).
`:fullscreen`	Selecciona si la página está en modo de pantalla completa.
`:host`	Hace referencia al elemento raíz (padre) de un componente con Shadow DOM.
`:indeterminate`	Selecciona el elemento cuando la casilla está en un estado indeterminado.
`:read-only`	Selecciona cuando el campo es de sólo lectura.
`:read-write`	Selecciona cuando el campo es editable por el usuario.
`:placeholder-shown`	Selecciona cuando el campo está mostrando un placeholder.
`:default`	Selecciona cuando el elemento tiene el valor por defecto.
`:required`	Cuando el campo es obligatorio, o sea, tiene el atributo `required`.
`:optional`	Cuando el campo es opcional (por defecto, todos los campos).
`:valid`	Cuando los campos cumplen la validación HTML5.
`:invalid`	Cuando los campos no cumplen la validación HTML5.
`:user-valid`	Igual que `:valid`, pero cuando el usuario ha interactuado.
`:user-invalid`	Igual que `:invalid`, pero cuando el usuario ha interactuado.
`:in-range`	Cuando los campos numéricos están dentro del rango.
`:out-of-range`	Cuando los campos numéricos están fuera del rango.
`:modal`	Selecciona el elemento que es de tipo modal[1].
`:any-link`	Selecciona un elemento que es un enlace.

[1] Una ventana o un elemento modal es aquel que centra la atención del usuario y no permite interacción con otros elementos que no son sus hijos.

4.5. Pseudoelementos

Los pseudoelementos son una característica de CSS que permite hacer referencias a "*comportamientos virtuales no tangibles*", es decir, seleccionar y dar estilo a elementos que no existen en el HTML o que no son un elemento en sí mismo.

La sintaxis de los pseudoelementos está precedida de **dos caracteres dos puntos** ('::') para diferenciarlos de las pseudoclases, que sólo tienen un carácter (:). No obstante, este cambio surgió después de su implementación, por lo que aún es frecuente ver pseudoelementos con la sintaxis de pseudoclase de un solo carácter dos puntos.

Vamos a explicar los 4 pseudoelementos más utilizados: `::before`, `::after`, `::first-letter` y `::first-line`. Hay algunos más, de reciente introducción, pero como aún no están completamente estandarizados no los expondremos aquí.

4.5.1. `::before` y `::after`

El pseudoelemento `::before` se refiere al primer hijo del elemento seleccionado, mientras que `::after` se refiere al último hijo. Ambos son comúnmente usados para añadir contenido cosmético a un elemento con la propiedad `content`. Es en línea (*inline*) de forma predeterminada.

Por ejemplo, digamos que queremos encerrar entre comillas cada vez que usemos el tag `<q>`:

```
q::before {content: "«";}
q::after  {content: "»";}
```

Entonces, el siguiente HTML producirá:

```
<q>Esto es una cita</q>
```

«Esto es una cita»

4.5.2. `::first-letter` y `::first-line`

Aunque `::before` y `::after` suelen ser los ejemplos de pseudoelementos más frecuentes, existen muchos otros. De entre los más utilizados podríamos encontrar:

- `::first-letter`. Aplica los estilos en la primera letra de un texto.
- `::first-line`. Aplica los estilos en la primera línea de un texto.

Por ejemplo:

```
<style>
  p::first-letter { font-size: 2rem; }
  p::first-line   { color: red; }
</style>

<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Nostrum,
doloremque.</p>
```

Obtenemos:

4.5.3. La propiedad `content`

La propiedad `content` se emplea para generar nuevo contenido de forma dinámica e insertarlo en la página. El estándar CSS 2.1 indica que la propiedad `content` sólo puede utilizarse en los pseudo-elementos `:before` y `:after`. Como su propio nombre indica, estos pseudo-elementos permiten seleccionar (y por tanto modificar) la parte anterior o posterior de un elemento de la página.

Como CSS es un lenguaje de hojas de estilos cuyo único propósito es controlar el aspecto o presentación de los contenidos, algunos diseñadores defienden que no es correcto generar nuevos contenidos mediante CSS.

Podemos insertar texto:

```
p:before {
  content: "Capítulo ";
}
```

O incluso una imagen:

```
.element::before {
  content: url("https://picsum.photos/100/100");
}
```

4.6. Webgrafía

- https://www.w3.org/TR/selectors-4/
- https://lenguajecss.com/css/selectores/selectores-basicos/
- https://developer.mozilla.org/es/docs/Web/CSS/CSS_Selectors
- https://www.w3schools.com/cssref/pr_gen_content.php
- https://www.w3.org/TR/selectors-4/

5. *Custom properties* (variables CSS)

Existe una forma nativa de guardar valores a través de *variables* CSS. Estas variables CSS no existían hace años y fue una de las razones por las que se popularizaron preprocesadores como LESS o Sass, que sí las incorporaban.

Las *custom properties* permiten dar un valor personalizado a las propiedades. El objetivo principal suele ser evitar escribir múltiples veces ese valor y, en su lugar, ponerle un nombre más lógico y fácil de recordar que hará referencia al valor real.

Así, si necesitamos cambiar el valor en algún momento podemos hacerlo en esa propiedad personalizada y no en múltiples partes del documento, lo que mejora notablemente la mantenibilidad del código.

Para **definir una *custom property*** usamos dos guiones ('--') previos al nombre que queramos utilizar. Además, debemos fijarnos en el elemento que definimos la variable, en este ejemplo la pseudoclase `:root`:

```
:root {
  --color-fondo: black;
}
```

La pseudoclase `:root` hace referencia al elemento raíz del documento, es decir, al elemento `<html>`. La diferencia de utilizar `html` o `:root` como selector es que este último tiene más especificidad CSS. Mientras que `html` tiene 001, `:root` tiene 010.

Una *custom properties* **podrá ser utilizada por el elemento donde se define y por todos sus descendientes**. Por tanto, al colocarla en `:root` estamos indicando que la variable estará definida para el ámbito de esa etiqueta `<html>` o cualquier elemento hijo, es decir, a todo el documento. Sin embargo, ya veremos que podemos aplicar estas variables sólo a partes concretas del DOM de nuestra página.

Las *custom properties* se suelen agrupar en las primeras líneas de un bloque CSS separándolas de otras propiedades estándar. Esto facilita la lectura del código.

Para **utilizar una *custom property*** hay que hacerlo dentro de la expresión `var()`:

```
.element {
  background: var(--color-fondo);
}
```

Es **muy recomendable** que la expresión `var()` tenga **dos parámetros**, siendo el segundo el valor por defecto en el caso de que esa propiedad no esté definida en el ámbito actual:

```
.element {
  background: var(--color-fondo, blue);
}
```

5.1. Ámbito de las *custom properties*

El ejemplo anterior es muy básico y puede que no se aprecie el detalle de los ámbitos con las *custom properties*, así que vamos a verlo con un ejemplo más específico.

Observa el siguiente ejemplo, también disponible en codepen, donde tenemos tres elementos con clase `child`, los dos primeros dentro de `parent` y el tercero fuera:

```
<div class="contenedor-verde">
  <div class="azul rojo">Primero</div>
  <div class="rojo">Segundo</div>
</div>
<div class="rojo">Tercero</div>

<style>
  .contenedor-verde { --color: green; }
  .azul             { --color: blue; }
  .rojo             { color: var(--color, red); }
</style>
```

La explicación es la siguiente:

- En el ámbito del primer `div` hay declarada dos veces la variable `--color`, una en `.contenedor-verde` y otra en `.azul`. Como el ámbito de `.primero` es más restrictivo que el de `.padre`, se toma el valor de esa variable, azul.
- En el ámbito del segundo `div` hay declarada solo una vez la variable `--color`, en `.padre`. Por tanto, se toma el valor de esa variable, verde en este caso.
- En el ámbito del tercer `div` no hay declarada ninguna variable `--color`. Por tanto, se toma el valor del segundo parámetro de `var()`, rojo en este caso.

5.2. *Custom properties* con JavaScript

Podemos añadir una *custom property* con JavaScript como si fuera una propiedad CSS cualquiera. Observa el siguiente ejemplo, disponible en codepen:

```
<div id="box">Hola mundo</div>
<button onclick="cambiaColor();">Cambiar el color</button>

<style>
  #box {
    font-size: 2rem;
    color: var(--color, blue);
  }
</style>
```

```
<script type="text/javascript">
  function cambiaColor() {
    document.getElementById("box").style.setProperty("--color", "red");
    // Valdría igual declarar la custom property en el elemento raíz,
    // lo único que cambia es su ámbito:
    // document.documentElement.style.setProperty("--color", "red");
    return;
  }
</script>
```

Dado que en el momento de renderizar la página no existe ninguna variable llamada `--color` en el ámbito de `#box`, el navegador utiliza el segundo parámetro de `var()` y toma el color azul. Al hacer clic en el botón, creamos la *custom property* en el ámbito de `#box` con el valor rojo, por lo que el color del texto se modifica.

Hemos declarado `--color` en el ámbito de `#box`, pero podríamos haberlo hecho en un ámbito superior, como el elemento raíz, y funcionaría exactamente igual.

```
document.documentElement.style.setProperty("--color", "red");
```

O con jQuery:

```
$("html").css("--color", "red");
```

Por supuesto, además de crear las propiedades, también podemos modificarlas. Observa el siguiente ejemplo, también disponible en codepen:

```
<div class="mover"></div>

<style>
  .mover {
    width: 50px;
    height: 50px;
    background: red;
    position: absolute;
    left: var(--mouse-x);
    top: var(--mouse-y);
  }
</style>

<script type="text/javascript">
  // Obtenemos el elemento raíz (<html>)
  let root = document.documentElement;

  // Notación clásica
  root.addEventListener("mousemove", function (e) {
    root.style.setProperty('--mouse-x', e.clientX + "px");
    root.style.setProperty('--mouse-y', e.clientY + "px");
  });
  // Con una expresión Lambda sería:
  // root.addEventListener("mousemove", e => {
  // });
</script>
```

6. At-rules (reglas @)

En CSS tenemos las denominadas *at-rules* (reglas precedidas del carácter '*@*'). Son un tipo de declaración especial que permite indicar comportamientos especiales en muchos contextos. Su sintaxis suele determinarse incluyendo una palabra clave que comienza por '@'.

Existen varias de estas *at-rules*, algunas en formato experimental y otras ya completamente soportadas por los navegadores. En esta sección solo presentaremos las 3 más utilizadas hasta la fecha. Existe una tercera también muy extendida, `@keyframes`, que veremos en el capítulo de transiciones y animaciones.

6.1. La regla `@import`

La regla `@import` permite cargar un fichero CSS externo e incorporarlo al archivo actual. Estas reglas CSS se suelen indicar en las primeras líneas del fichero, ya que deben figurar antes de otras reglas CSS o contenido CSS similar.

Existen **2 formas de importaciones directas** mediante `@import`: utilizando la función `url()` o indicando un texto con el archivo o dirección URL:

1. Importamos el fichero utilizando la función `url()`.

```
@import url("fichero.css");
```

2. Importamos el fichero utilizando un texto.

```
@import "fichero.css";
```

Se pueden utilizar los nombres de los archivos, rutas relativas o absolutas:

```
@import url("menu.css");              /* Fichero en la misma ruta */
@import url("menu/sidebar.css");      /* Ruta relativa */
@import "https://manz.dev/index.css"; /* Ruta absoluta */
```

Además de las importaciones directas también existen **importaciones condicionales**. Tenemos la posibilidad de indicar posteriormente a la URL una *media query*, que nos permitirá que esa hoja de estilos externa se descargue y procese sólo si estamos en un navegador que cumple las condiciones de la *media query*.

Observa cómo podemos indicar una *media query* para que ese archivo CSS se importe sólo en el caso de que se cumpla:

```
@import url("mobile.css") screen and (max-width: 640px);
@import url("print.css") print;
```

En el primer caso, el archivo mobile.css se descargará sólo si se está utilizando una pantalla que tenga como máximo 640px de ancho, presumiblemente un dispositivo móvil. En el segundo caso, el archivo print.css se aplicará sólo si estamos imprimiendo con el navegador la página actual, de lo contrario, no se descargará ni se aplicará.

También podemos combinar la regla `@supports` con la regla `@import` y establecer condiciones específicas, que veremos en el siguiente apartado.

La principal **desventaja** de usar `@import` es el rendimiento, dado que los navegadores cargan las hojas de estilo más rápido cuando se usa `<link>`. Además, versiones antiguas de navegadores podrían no soportar la regla `@import`.

Entre las **ventajas** de `@import` tenemos:

- Modularidad: Facilita la organización de archivos CSS en módulos, especialmente si tienes estilos base comunes en varios proyectos o componentes que quieres importar en archivos secundarios sin modificar el HTML.
- CSS condicional: Como hemos visto, podemos usar `@import` dentro de *media queries*, permitiendo cargar archivos CSS solo bajo ciertas condiciones, por ejemplo:
- Integración con preprocesadores: Herramientas como SASS o PostCSS usan `@import` para combinar múltiples archivos CSS en uno solo durante el proceso de compilación, lo que mejora la organización en el desarrollo sin impactar el rendimiento en producción.

6.2. La regla `@supports`

La regla `@supports` permite establecer fragmentos de código CSS condicionales, aplicando estilos CSS sólo cuando se cumplen ciertas condiciones. Esto puede ser muy útil para aplicar unos estilos si el navegador soporta una característica, o aplicar un estilo diferente como *fallback* si no lo hace.

Un ejemplo de la regla `@supports` podría ser la siguiente:

```
@supports (display: grid) {
  .content {
    display: grid;
    grid-template-columns: 1fr 1fr;
  }
}
```

Definimos ciertos estilos para la clase `.content` que el navegador sólo aplicará en el caso de que tenga soporte para la propiedad `display` con el valor `grid`. Podemos hacer esto mismo con cualquier otra propiedad CSS y utilizar la regla `@supports` para crear códigos condicionales.

Podemos crear también reglas compuestas un poco más complejas. Por ejemplo, combinemos una regla de negación con una normal:

```
@supports not (display: grid) and (display: flex) {
  .content {
    display: flex;
    justify-content: center;
  }
}
```

En este ejemplo creamos una clase `.content` con contenido estructurado con flex, siempre en el caso de que el navegador no soporte grid pero sí flex. Observa que hemos combinado tanto el `not`, que afecta sólo a la primera condición, como el `and` que afecta a ambas y exige que se cumplan ambas.

Si quisiéramos crear una doble condición con ambas negadas, deberíamos hacer similar a este ejemplo:

```
@supports (not (display: flex)) and (not (display: grid)) {
  .box {
    display: inline-block;
  }
}
```

Esto podría ser interesante, pero recuerda no utilizarlo con propiedades muy antiguas. La regla `@supports` fue implementada en navegadores alrededor del año 2019-2020, por lo que utilizarla para excluir navegadores muy antiguos no funcionará porque tampoco tendrán la regla `@supports` implementada.

En lugar de establecer reglas negadas compuestas, es mejor utilizar un enfoque donde establezcas unos estilos generales, que se sobrescriban si el navegador tiene implementadas nuevas características, como explicaremos a continuación.

En el siguiente ejemplo verás un primer bloque de código fuera de reglas, que se aplicará en cualquier navegador, moderno o antiguo. Sin embargo, a continuación, tenemos dos reglas `@supports` que se ejecutarán en navegadores más actuales:

```
.content {
  display: inline-block;
}

@supports (display: grid) {
  .content {
    display: grid;
    grid-template-columns: 1fr;
    justify-content: center;
  }
}

@supports (not (display: grid)) and (display: flex) {
  .content {
    display: flex;
    justify-content: center;
  }
}
```

En el caso de tratarse de un navegador que implemente grid, este establecerá los estilos indicados en el primer bloque con la regla `@supports`. Luego, la siguiente regla `@supports` se ejecutará sólo en el caso de que el navegador no soporte grid pero sí flex.

6.3. La regla `@media` (*media queries*)

La regla `@media`, usada para definir *media queries*, es un tipo de regla de CSS que permite crear un bloque de código que sólo se procesará cuando se cumplan los criterios especificados como condición.

```
@media screen and (*condición*) {
  /* reglas CSS */
}

@media screen and not (*condición*) {
  /* reglas CSS */
}
```

Con este método, especificamos que queremos aplicar los estilos CSS para tipos de medios concretos (*screen*: sólo en pantallas, en este caso) que cumplan las condiciones especificadas entre paréntesis. De esta forma, una estrategia aconsejable es crear reglas CSS generales aplicadas a todo el documento: colores, tipo de fuente… y luego las particularidades que se aplicarían sólo en el dispositivo en cuestión.

Aunque suele ser menos habitual, también se pueden indicar reglas `@media` negadas mediante la palabra clave `not`, que aplicará CSS siempre y cuando no se cumpla una determinada condición. También pueden separarse por comas varias condiciones de *media queries*.

Al igual que `not`, también existe una palabra clave `only` que, suele usarse a modo de *hack*. El comportamiento por defecto ya incluye los dispositivos que encajan con la condición, pero con la palabra clave `only` conseguimos que navegadores antiguos que no la entienden, no procesen la información, dejándola sólo para navegadores modernos.

Existen los siguientes **tipos de medios**:

- **screen**. Pantallas. Es el más común.
- **print**. Versiones de impresión o pantallas de previsualización de impresión.
- **speech**. Lectores de texto para invidentes.
- **all**. Todos los dispositivos o medios. El que se utiliza por defecto.

Recordemos que con el siguiente fragmento de código HTML estamos indicando que el nuevo ancho de la pantalla es el ancho del dispositivo, por lo que el aspecto del viewport se va a adaptar consecuentemente:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Con esto conseguiremos preparar nuestra web para dispositivos móviles y prepararnos para la introducción de *media query* en el documento CSS.

Veamos un ejemplo clásico de *media query*. Observa que en el código existen 3 bloques `@media` donde se definen estilos CSS para cada uno de esos tipos de dispositivos.

```css
@media screen and (max-width: 640px) {
  .menu { background-color: blue; }
}

@media screen and (min-width: 640px) and (max-width: 1280px) {
  .menu { background-color: red; }
}

@media screen and (min-width: 1280px) {
  .menu { background-color: green; }
}
```

El ejemplo muestra un elemento (clase `menu`) con un color de fondo concreto dependiendo del tipo de medio con el que se visualice la página:

- Azul para resoluciones menores a 640 píxeles de ancho (móviles).
- Rojo para resoluciones entre 640 píxeles y 1280 píxeles de ancho (tablets).
- Verde para resoluciones mayores a 1280 píxeles (desktop).

El número de bloques de reglas `@media` que se utilicen depende del desarrollador web, ya que no es obligatorio utilizar un número concreto. Se pueden utilizar desde un sólo *media query*, hasta múltiples de ellos a lo largo de todo el documento CSS.

Hay que tener en cuenta que las *media queries* también es posible indicarlos desde HTML, utilizando la etiqueta <link>:

```html
<link rel="stylesheet" href="mobile.css" media="screen and (max-width: 640px)">
<link rel="stylesheet" href="tablet.css" media="screen and (min-width: 640px) and (max-width: 1280px)">
<link rel="stylesheet" href="desktop.css" media="screen and (min-width: 1280px)">
```

Estos estilos quedarán separados en varios archivos diferentes. Ten en cuenta que todos serán descargados al cargar la página, sólo que no serán aplicados al documento hasta que cumplan los requisitos indicados en el atributo media.

En los ejemplos anteriores solo hemos utilizado `max-width` y `min-width` como **tipos de características** a utilizar en condiciones de *media query*. Sin embargo, tenemos una lista de tipos de características que podemos utilizar:

- **width**. Si el dispositivo tiene el tamaño indicado exactamente. No se usa.
- **min-width**. Si el dispositivo tiene un tamaño de ancho mayor al indicado.
- **max-width**. Si el dispositivo tiene un tamaño de ancho menor al indicado.
- **aspect-ratio**. Si el dispositivo encaja con la proporción de aspecto indicada.
- **orientation**. Si el dispositivo está en colocado en modo vertical o apaisado. Los posibles valores son `landscape` o `portrait`.

A partir de XXX se introdujeron ciertos cambios en las media queries de modoque se empezaron a poder usar rangos

Comentado [IE1]: HACER

6.4. La regla `@container` (*container queries*)

Como hemos visto en el apartado anterior, las *media queries* permiten para dar estilo a elementos dependiendo de si se cumple una cierta condición, como el tamaño, orientación o cierta característica de la página o dispositivo en el que nos encontramos. Es uno de los mecanismos principales del *responsive design*.

Las ***container queries*** siguen el mismo concepto de las *media queries*, pero en lugar de modificar los estilos dependiendo del tamaño de la página o dispositivo, **modifica los estilos dependiendo de un contenedor padre o ancestro específico**. De esta forma, podemos cambiar el tamaño de ciertos elementos y hacer que tengan una forma o unos estilos concretos dependiendo del contexto donde se encuentren.

Hay dos tipos de *container queries*, ***container size queries*** and ***container style queries*** (consultas de tamaño de contenedor y consultas de estilo de contenedor):

- ***Container size queries***: Aplican estilos a elementos en función del tamaño de un elemento contenedor (también orientación y relación de aspecto). Los elementos contenedores deben declararse explícitamente como *container size queries*.

- ***Container style queries:*** Aplican estilos a elementos en función de las características de estilo de un elemento contenedor. Cualquier elemento que no esté vacío puede ser un contenedor de consultas de estilo. Actualmente, la única característica de estilo compatible con las consultas de estilo son las propiedades personalizadas de CSS. En este caso, la consulta devuelve verdadero o falso según el valor calculado de las propiedades personalizadas del elemento contenedor. Cuando las consultas de estilo de contenedor sean totalmente compatibles, permitirán aplicar estilos a los descendientes de cualquier elemento en función de cualquier propiedad, declaración o valor calculado; por ejemplo, si el contenedor es `display: inline` o tiene un color de fondo no transparente.

Actualmente (septiembre de 2024) las *container size queries* tienen una compatibilidad del 90% y las *container style queries* del 71%, según caniuse.

6.4.1. Container size queries

Como hemos dicho, las *container size queries* permiten aplicar estilos a elementos en función del tamaño de un elemento contenedor (también orientación y relación de aspecto). Para declarar a un elemento como contenedor de consulta de tamaño debemos establecer la propiedad `container-type` (o la abreviatura `container`) en `size` o `inline-size`.

Veamos un ejemplo, disponible en codepen.

La propiedad `container-name` establece un nombre al contenedor que hace que la condición del contenedor sea más específica. Se evaluará sólo para los elementos que tengan ese nombre establecido en la propiedad de nombre de contenedor.

Veamos un ejemplo, disponible en codepen.

Un aspecto **importante** a tener en cuenta es que cuando ponemos `width` o `height` como condición de una *container query*, estos valores se toman del `content-box` del elemento, sea cual sea el valor de la propiedad `box-sizing`.

Para usar *container queries* primero necesitaremos declarar un *containment context* en un elemento, de esta manera el navegador sabrá que quieres consultar las dimensiones de ese contenedor. Para hacer esto, usa la propiedad `container-type` sobre el elemento que quieres que actúe como contenedor. Los posibles valores son:

- `size`: El elemento responderá tanto a su tamaño en línea como en bloque. Esto permite que las consultas respondan tanto a la anchura como a la altura del contenedor.
- `inline-size`: El elemento responderá a su tamaño en línea (la dimensión horizontal, la anchura, en la mayoría de los idiomas). Es el más usado.
- `normal`: El elemento no es un *query container* para consultas de tamaño.

Para empezar, tenemos que determinar cuál será el elemento contenedor al que vamos a hacer referencia. En ese elemento necesitaremos establecer las siguientes propiedades:

Propiedad	Valor	Descripción

`container-name`	*none* | nombre del contenedor	Establece un nombre de contenedor para poder hacer referencia a él.
`container-type`	*normal* | *size* | *inline-size*	Establece el tipo de tamaño de contenedores (bloque, en línea…).

La propiedad `container-name` le da un nombre de contenedor al elemento en el que nos encontramos.

La propiedad `container-type` establece el tipo de tamaño que va a tener el contenedor:

- *size* para elementos de tipo bloque.
- *inline-size* para elementos de tipo *inline*.
- *inline-size*, no tendrá en cuenta la altura, si la colocamos en la regla `@container`, al contrario que si usamos *size*.

Ejemplo en codepen.

6.5. Webgrafía

- https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule
- https://lenguajecss.com/css/reglas-css/que-son-reglas-css/
- https://www.w3schools.com/css/css_rwd_mediaqueries.asp
- https://www.w3.org/TR/css-contain-3/#container-queries
- https://developer.mozilla.org/es/docs/Web/CSS/CSS_containment/Container_queries
- https://lenguajecss.com/css/responsive-web-design/css-container-queries/

7. Anidamiento

La idea detrás del concepto de CSS Nesting (CSS anidado) es tener fragmentos o bloques de código unos dentro de otros, haciendo que estos selectores sean mucho más intuitivos para el programador y más fáciles de mantener.

Veamos un ejemplo muy sencillo. Con CSS tradicional tendríamos:

```
.container {
  background: grey;
}

.container .item {
  background: orangered;
}
```

En este código se ve a simple vista que el segundo selector hace referencia a todos los elementos con clase `.item` que sean descendientes de elementos con clase `.container`. Sin embargo, cuando el código crece esto se puede complicar. Y si, además, existen colisiones de nombres de selectores, aún peor.

Con anidamiento, el código equivalente sería el siguiente:

```
.container {
  background: grey;

  .item {
    background: orangered;
  }
}
```

En este caso, el selector `.item` está ubicado en el interior del selector `.container`, por lo que el código queda más claro, más organizado y es más fácil de mantener.

Hay que resaltar que, en marzo de 2024, el **CSS Nesting no forma parte del estándar de la W3C**. Su estado es aún Editor's Draft, lo que indica que se trata de un trabajo en progreso con intenciones de ser estándar pero aún no publicado para su revisión, esperando que el continuo aporte de los miembros involucrados en esta especificación logre los requisitos técnicos suficientes para pasar a la siguiente etapa.

Sin embargo, los navegadores ya están empezando a dar soporte a esta característica, que era una de las más demandadas por la comunidad. Puedes ver el soporte de los navegadores en la página de caniuse.

Para más información sobre esto puedes consultar la página del módulo.

8. Imágenes *responsive*, *layout shift* y `aspecto-ratio`

Ya deberíamos saber que si queremos que nuestras imágenes se adapten a diferentes anchuras de pantalla debemos indicarlo con el siguiente CSS:

```css
img {
  width: 100%;
  height: auto;
}
```

O, si queremos que la imagen se reduzca si es necesario, pero nunca se amplíe para que sea mayor que su tamaño original:

```css
img {
  max-width: 100%;
  height: auto;
}
```

Yendo más allá, podemos establecer una anchura preferida, pero manteniendo el comportamiento responsive:

```css
img {
  width: 100%;
  max-width: 400px;
  height: auto;
}
```

También es frecuente incluir `display: block` para asegurar que nada se muestra en línea con la imagen. Carga el siguiente código en un editor para ver el comportamiento *responsive*:

```html
<style>
  img {
    width: 100%;
    max-width: 400px;
    height: auto;
    display: block;
  }
</style>

<img src="https://picsum.photos/id/684/600/400" alt="Imagen de un gato">

<p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Id voluptate aut necessitatibus obcaecati doloribus, dolorum molestiae cum sit corporis.</p>
```

Esto funciona perfectamente. Sin embargo, existe un inconveniente cuando el cliente dispone de una conexión lenta. Hablamos del ***Cumulative Layout Shift* (CLS)**.

8.1. *Cumulative Layout Shift* (CLS)

El *Cumulative Layout Shift* es una métrica utilizada para evaluar la estabilidad visual de una página web durante su carga. Se trata de una métrica importante dentro del conjunto de métricas de Experiencia del Usuario (UX), especialmente en lo que respecta a la percepción de los usuarios sobre la estabilidad y la fluidez de la página mientras navegan por ella.

CLS se centra en los desplazamientos inesperados del contenido visual dentro de una página web mientras esta se carga. Estos desplazamientos pueden ser causados por varios factores, como la carga retardada de elementos de la página, cambios en el diseño debido a la carga de contenido dinámico (como anuncios, imágenes o scripts) o la ejecución tardía de ciertos estilos CSS.

Para calcular el CLS de una página web, se tienen en cuenta dos factores:

- **Desplazamiento visible**: Se refiere al cambio en la posición de cualquier elemento visual en la página web, que sea perceptible para el usuario. Esto incluye elementos como texto, imágenes, botones, etc.
- **Área de impacto**: Se refiere al área afectada por el desplazamiento. Cuanto más grande sea el área afectada, mayor será el impacto del desplazamiento en la percepción del usuario.

El CLS se calcula multiplicando el desplazamiento visible por el área de impacto para cada cambio que ocurra en la página durante su carga. Luego, estos valores se suman para obtener el CLS total.

Una buena práctica es mantener el CLS tan cercano a cero como sea posible, lo que indica una experiencia de usuario más estable y sin interrupciones. Para lograr esto, los desarrolladores web pueden tomar varias medidas, como:

- Asignar dimensiones a los elementos multimedia (imágenes, videos) para evitar cambios repentinos en su tamaño durante la carga.
- Reservar espacio para anuncios y contenido dinámico para evitar que desplacen el contenido principal de la página.
- Priorizar la carga de elementos críticos y evitar la ejecución de scripts que puedan alterar el diseño de la página después de que se haya renderizado inicialmente.

¿Por qué estamos hablando de esto en imágenes responsive? Pues porque la carga de imágenes es una de las responsables del aumento del CLS.

Imagina el siguiente escenario, **incluimos una imagen sin especificar el tamaño exacto y esta se dibuja después de que se haya renderizado el resto del contenido de la página**. Esto causará un desplazamiento repentino del layout. Es decir, un *layout shift*.

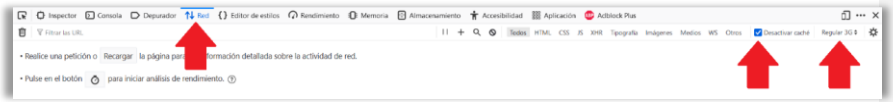
Usa el código que vimos antes para hacer la prueba:

```
<style>
  img {
    width: 100%;
    max-width: 400px;
    height: auto;
    display: block;
  }
</style>

<img src="https://picsum.photos/id/684/600/400" alt="Imagen de un gato">

<p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Id voluptate aut
necessitatibus obcaecati doloribus, dolorum molestiae cum sit corporis.</p>
```

Podemos simular una conexión lenta con las herramientas de desarrollador:



Si ahora cargamos de nuevo la pagina veremos el *Layout Shift* en acción. Para solucionar esto tenemos dos opciones, como hemos visto:

- Indicamos directamente las dimensiones de la imagen con `width` y `height`.
- Indicamos una de esas propiedades (habitualmente `width`) y `aspecto-ratio`.

Si lo hacemos con `width` y `height` tenemos el problema de que la imagen dejará de ser *responsive*. Así que solo nos queda la segunda opción, que es usar `aspecto-ratio`:

```
img {
  width: 100%;
  max-width: 400px;
  height: auto;
  display: block;
  aspect-ratio: 3/2;
}
```

Ahora nuestra imagen es completamente responsive y hemos eliminado el *layout shift* con una sola línea de CSS.

Puede ocurrir que al establecer el `aspecto-ratio` distorsionemos la imagen si este valor no es el original de la imagen. Por ejemplo, la imagen que hemos usado en el ejemplo tiene unas dimensiones originales de 1200x800, lo que implica una relación de aspecto de 3/2. Si establecemos, por exagerar, 21/9, observarás la distorsión. Para evitar esta distorsión podemos usar la propiedad `object-fit`, que indica al navegador cómo redimensionar la imagen para ajustarse al contenedor.

9. Transiciones y animaciones

En CSS aparecen uno de los aspectos más interesantes de una web interactiva: las animaciones y transiciones. Hasta ahora, al utilizar una pseudoclase como `:hover`, el cambio de estilos ocurría de golpe, pasando de un estado inicial a otro final. Ahora tendremos a nuestra disposición flexibilidad para dotar de atractivos efectos de transición que harán que nuestros diseños sean más elegantes.

9.1. Transiciones. La propiedad `transition`

Las propiedades CSS que podemos utilizar relacionadas con las transiciones son:

Unidad	Valor	Descripción
`transition-property`	`all | none | propiedad`	Propiedades afectadas por la transición.
`transition-duration`	`0 | tiempo`	Tiempo de duración.
`transition-timing-function`	`Función de tiempo`	Ritmo de la transición.
`transition-delay`	`0 | tiempo`	Tiempo de retardo inicial.
`transition`		Abreviatura de las 4 anteriores, en el orden de aparición.

No todas las propiedades CSS se pueden animar. En general, se pueden animar la mayoría de las propiedades cuantificables (valores numéricos, porcentajes, valores hexadecimales como colores…).

Veamos un ejemplo, disponible en w3schools:

```
div {
  width: 100px;
  height: 100px;
  background: red;
  transition-property: width;
  transition-duration: 2s;
  transition-timing-function: linear;
  transition-delay: 1s;
}

div:hover {
  width: 300px;
}
```

Estas 4 propiedades podríamos haberlas resumido en:

```
transition: width 2s linear 1s;
```

Que tiene el siguiente formato:

```
transition: <property> <duration> <timing-function> <delay>
```

Podríamos eliminar el *delay* de 1s de la expresión anterior:

```
transition: width 2s linear;
```

O el retardo y la función (linear):

```
transition: width 2s;
```

O dejar el retardo y quitar la función:

```
transition: width 2s 1s;
```

Lo que no podemos quitar es la duración, porque por defecto es cero.

Resumiendo, para 1 propiedad:

```
/* nombre de la propiedad | duración */
transition: width 4s;

/* nombre de la propiedad | duración | retardo */
transition: width 4s 1s;

/* nombre de la propiedad | duración | función */
transition: width 4s ease-in-out;

/* nombre de la propiedad | duración | función | retardo */
transition: width 4s ease-in-out 1s;
```

Si queremos animar más propiedades, separamos por comas:

```
transition: width 4s, height 1s;
```

Con respecto a las **funciones de tiempo** tenemos dos propiedades que se aplican exactamente igual y se encargan de definir el ritmo o transcurso de una animación o una transición. Son `transition-timing-function` y `animation-timing-function`.

Existen una serie de palabras reservadas que definen ciertas funciones de tiempo. Cada una de ellas realiza la animación a un ritmo diferente:

Valor	Inicio	Transcurso	Final
ease	Lento	Rápido	Lento
linear	Normal	Normal	Normal
ease-in	Lento	Normal	Normal
ease-out	Normal	Normal	Lento
ease-in-out	Lento	Normal	Lento

Además de usar `:hover`, podemos aplicar transiciones de varias formas más:

- Otros pseudoelementos: Podemos usar `:focus`, `:active` o `:checked`.
- Transiciones automáticas: Podemos aplicar transiciones directamente al cambiar el estilo de un elemento mediante JavaScript, por ejemplo, modificando su clase con `classList.toggle()`.
- Transiciones en el `:root` o `body`: Podemos aplicar transiciones a propiedades definidas en *custom properties* en el `:root`, permitiendo que afecten a múltiples elementos cuando cambian los valores de estas variables.

Tienes un ejemplo de estas opciones en el siguiente codepen.

Un último ejemplo. Queremos que el color de fondo de un párrafo haga una transición a "coral" en 0.5 segundos utilizando la función *ease-in-out* cuando el ratón pase por encima. Sería:

```
<p>Lorem ipsum dolor sit</p>
<style>
  p { transition: background-color 0.5s ease-in-out; }
  p:hover { background-color: coral; }
</style>
```

Veamos algunos ejercicios:

1. Crea una transición de 1 segundo en la que una caja cuadrada de 100x100 píxeles pase a ocupar 300x300 al pasar el ratón por encima. Usa la función de tiempo `ease-in-out` y un retardo de 0.5 segundos.
2. Modifica la transición anterior para que las funciones de tiempo de `height` y `width` sean `ease-in` y `ease-out`.
3. Crea una transición que haga que una caja cuadrada cambie su opacidad suavemente cuando el usuario haga clic en ella.
4. Crea una transición que haga que una caja cuadrada cambie su color de fondo suavemente de rojo a verde cuando el usuario pase el cursor sobre ella.

Aquí tienes las soluciones:

- Ejercicio 1.
- Ejercicio 2.
- Ejercicio 3.
- Ejercicio 4.

9.2. Animaciones. La propiedad `animation` y `@keyframes`

Una vez conocemos las transiciones es muy fácil adaptarnos al concepto de animación, que amplia lo que ya sabemos de transiciones convirtiéndolo en algo mucho más flexible y potente. Ya no será necesario que el usuario interactúe de alguna forma como pasa en las transiciones.

Las **transiciones** suavizan un cambio de un estado inicial a un estado final. Las **animaciones** parten del mismo concepto, pero permiten añadir estados intermedio. Así pues, con las animaciones podemos partir desde un estado inicial, a un estado posterior, a otro estado posterior, y así sucesivamente.

Para crear animaciones CSS es necesario realizar **2 pasos**:

1. Indicar qué elemento HTML vamos a animar mediante la propiedad `animation` (o derivadas).
2. Definir la animación y sus estados (fotogramas) mediante la regla `@keyframes`.

Primero vamos a examinar las diferentes propiedades relacionadas con las animaciones y más adelante veremos cómo crear fotogramas con la regla `@keyframes`.

Propiedad	Valor	Descripción
`animation-name`	`none` | *nombre*	Nombre de la animación.
`animation-duration`	`0` | tiempo	Duración de la animación.
`animation-timing-function`	*función de tiempo*	Ritmo de la animación.
`animation-delay`	`0` | tiempo	Retardo en iniciar la animación.
`animation-iteration-count`	`1` | infinite | número	Número de veces que se repetirá.

`animation-direction`	`normal`
	`reverse`
	`alternate`
	`alternate-reverse`
		Establece si una animación debe reproducirse hacia adelante, hacia atrás o alternar entre hacia adelante y hacia atrás.
		Demo en mozilla.org.

`animation-fill-mode`	`none`
	`forwards`
	`backwards`
	`both`
		Establece cómo una animación aplica estilos antes y después de su ejecución.
		Demo en mozilla.org.

`animation-play-state`	`running` | `paused`	Estado de la animación.

Las propiedades `animation-duration` y `animation-delay` funcionan exactamente igual que las propiedades análogas `transition-duration` y `transition-delay` del apartado de transiciones. De igual forma, la propiedad `animation-timing-function` es idéntica a la propiedad `transition-timing-function` que explicamos anteriormente.

Quizá la propiedad `animation-fill-mode` sea la menos intuitiva de todas. Por defecto, una animación antes de arrancar y después de terminar (si no está establecida en repetición infinite) no tiene aplicados los estilos de la animación especificada. Esto se puede ver fácilmente cuando termina una animación, que vuelve a sus estilos iniciales. Mediante la propiedad `animation-fill-mode` podemos indicar qué debe hacer la animación cuando no se está reproduciendo:

- El valor `none` realiza el comportamiento indicado en el párrafo anterior.
- El valor `backwards` indica que, antes de empezar, la animación debe tener aplicados los estilos del fotograma inicial.
- El valor `forwards` indica que, al terminar, la animación debe tener aplicados los estilos del fotograma final.
- El valor `both` indica que se deben aplicar los dos casos anteriores (`backwards` y `forwards`).

Por último, la propiedad `animation-play-state` permite establecer la animación a estado de reproducción `running` o pausarla mediante el valor `paused`. Esto en CSS no da demasiadas posibilidades, pero puede ser muy útil combinado con JavaScript.

Nuevamente, CSS ofrece la posibilidad de resumir todas estas propiedades en una sola. El orden recomendado para los valores de la propiedad de atajo sería:

```
animation: <name> <duration> <timing-function> <delay> <iteration-count>
<direction> <fill-mode> <play-state>
```

Por ejemplo:

```
animation: change-color 5s linear 0.5s 4 normal forwards running;
```

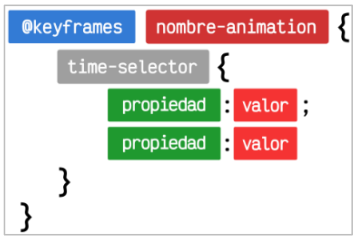
Como en el caso de las transiciones, es posible separar por comas para indicar que queremos realizar varias animaciones a la vez. Si además añadimos la propiedad `animation-delay`, podemos hacer algo muy interesante: **encadenar animaciones**.

Observa el siguiente ejemplo donde se verá mucho mejor:

```
animation:
  move-right 5s linear 0s,   /* Empieza a los 0s (no hay anterior) */
  look-up 2.5s linear 5s,    /* Empieza a los 5s (5 de la anterior) */
  move-left 5s linear 7.5s,  /* Empieza a los 7.5s (5 + 2.5) */
  dissapear 2s linear 12.5s; /* Empieza a los 12.5s (5 + 2.5 + 5) */
```

Hemos dejado para el final del capítulo la **creación de los fotogramas**. Una animación está formada por varios fotogramas, una secuencia de imágenes que generan el efecto de movimiento que conocemos de una animación. En CSS, los fotogramas se crean a partir de propiedades CSS.

Para definir esos fotogramas clave utilizaremos la *at-rule* `@keyframes`, que se basa en el siguiente esquema:



Cada uno de estos *time-selector* será un momento clave de cada uno de los fotogramas clave de nuestra animación. Por ejemplo:

```
@keyframes change-color {
  /* Primer fotograma */
  from { background: red; }
  /* Segundo y último fotograma */
  to { background: green; }
}
```

Las palabras clase `from` y `to` se refieren a un momento específico de la animación y equivalen a 0% y 100%, inicio y final de la animación.

Hemos nombrado la animación como `change-color`, que parte de un primer fotograma clave con el fondo rojo hasta un último fotograma clave con fondo verde.

Una vez definida la animación, hay que asociarla al elemento que queramos animar. Por ejemplo:

```
.animated {
  background: grey;
  color: #FFF;
  width: 150px;
  height: 150px;
  animation: change-color 2s ease 0s infinite;
}
```

Y ya está, ya tenemos hecha nuestra animación. Pero podemos hacer más. Como hemos dicho, las animaciones permiten estados intermedios, así que vamos a añadir un fotograma intermedio. Puedes ver este **ejemplo 1** completo en codepen.

```
@keyframes change-color {
  0% {
    background: red;    /* Primer fotograma */
  }
  50% {
    background: yellow; /* Segundo fotograma */
    width: 400px;
  }
  100% {
    background: green;  /* Último fotograma */
  }
}
```

Veamos algunos ejemplos más.

Ejemplo 2. Este es un ejemplo que ilustra de los distintos valores de la propiedad `animation-direction`. También disponible en codepen.

```
<div class="normal">normal</div>
<div class="reverse">reverse</div>
<div class="alternate">alternate</div>
<div class="alternate-reverse">alternate-reverse</div>

<style>
  div {
    display: grid;
    justify-items: center;
    align-items: center;
    width: 100px;
    height: 100px;
    background: red;
    position: relative;
    margin-bottom: 10px;
    animation: my-animation 4s 2;
  }
  div.normal { animation-direction: normal; }
  div.reverse { animation-direction: reverse; }
  div.alternate { animation-direction: alternate; }
  div.alternate-reverse { animation-direction: alternate-reverse; }

  @keyframes my-animation {
    0% {
      background: red;
      left: 0px;
    }

    25% {
      background: yellow;
      left: 100px;
    }

    50% {
      background: blue;
      left: 200px;
    }

    75% {
      background: green;
      left: 300px;
    }

    100% {
      background: red;
      left: 400px;
    }
  }
</style>
```

Ejemplo 3. En este otro ejemplo ilustramos de los distintos valores de la propiedad `animation-fill-mode`. También disponible en codepen.

```
<style>
  div {
    display: grid;
    justify-items: center;
    align-items: center;
    width: 100px;
    height: 100px;
    background: grey;
    position: relative;
    left: 0px;
    margin-bottom: 10px;
    animation: my-animation 4s;
    animation-delay: 2s;
  }

  div.none { animation-fill-mode: none; }
  div.forwards { animation-fill-mode: forwards; }
  div.backwards { animation-fill-mode: backwards; }
  div.both { animation-fill-mode: both; }

  @keyframes my-animation {
    0% {
      background: red;
      left: 100px;
    }

    25% {
      background: yellow;
      left: 200px;
    }

    50% {
      background: blue;
      left: 300px;
    }

    75% {
      background: green;
      left: 400px;
    }

    100% {
      background: red;
      left: 500px;
    }
  }
</style>
```

Ejercicio 1. Crea una animación para un botón al pasar el ratón por encima. Usa el siguiente código inicial:

```
<button class="hover-button">Enviar</button>

<style>
  .hover-button {
    padding: 0.5em 1em;
    background-color: black;
    color: white;
    border: 2px solid black;
    cursor: pointer;
    font-size: 4rem;
  }
</style>
```

La animación deberá tener las siguientes características:

- Durará 2 segundos, sin retardo.
- Durante los primeros 0.2 segundos de la animación, el botón pasará a tener un tamaño de letra un 50% superior a su estado inicial.
- Desde ese momento hasta el final, el botón alternará sus colores de fondo y texto, de fondo negro y texto blanco a fondo blanco y texto negro dos veces.

Solución en codepen.

Ejercicio 2. Modifica el ejercicio anterior para conseguir que el paso de fondo negro y texto blanco a fondo blanco y texto negro se produzca indefinidamente.

Solución en codepen.

Ejercicio 3. Crea una animación para que un elemento cuadrado haga lo siguiente:

1. Aumente su `width` al doble durante 1 segundo.
2. Aumente su `height` al doble durante 1 segundo.
3. Reduzca su `width` a su estado original en 0.5 segundos.
4. Reduzca su `height` a su estado original en 0.5 segundos.

Todos estos pasos deben ir encadenados, cuando acabe uno comienza el otro. Además, la animación será infinita.

Solución en codepen.

9.3. Webgrafía

- https://www.w3schools.com/css/css3_animations.asp
- https://www.w3schools.com/css/css3_transitions.asp
- https://lenguajecss.com/css/animaciones/transiciones/
- https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Transitions/Using_CSS_transitions

10. Transformaciones

Las transformaciones son una de las características de CSS más interesantes y potentes para convertir las hojas de estilo en un sistema capaz de realizar efectos visuales 2D y 3D. Con ellas podemos hacer cosas como mover elementos, rotarlos, aumentarlos o disminuirlos y otras transformaciones relacionadas o combinadas.

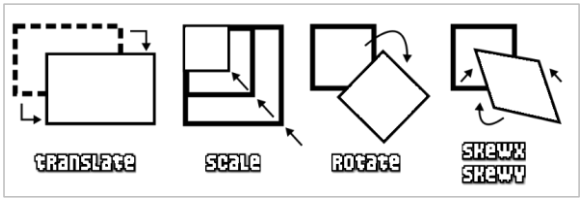
10.1. La propiedad *transform*

Esta propiedad aplica una (o varias) funciones de transformación al elemento al que se aplica. Su sintaxis básica es:

```
transform: funcion1, funcion2, funcion3...
```

Las funciones de transformación que podemos aplicar son:

- **Translación.** Desplaza un elemento en el eje X (izquierda, derecha) y/o en el eje Y (arriba, abajo).
- **Escalado.** Escala el elemento una determinada cantidad más grande o más pequeña. También se puede voltear.
- **Rotación.** Gira el elemento sobre su eje X o sobre su eje Y. También se puede girar sobre sí mismo.
- **Deformación.** Inclina el elemento sobre su eje X o sobre su eje Y.



10.2. Funciones de translación

Entre las funciones de translación, podemos encontrar:

Función	Descripción
`translateX(x)`	Traslada el elemento una distancia 'x' horizontalmente.
`translateY(y)`	Traslada el elemento una distancia 'y' verticalmente.
`translate(x, y)`	Propiedad atajo de las dos anteriores.
`translate(x)`	Equivalente a `translate(x, 0)`.
`translateZ(z)`	Traslada el elemento en el eje z.
`translate3d(x, y, z)`	Traslada el elemento en los tres ejes.

Por ejemplo, imagina que queremos mover 100 píxeles a la derecha y 25 píxeles hacia abajo desde su posición normal en el flujo del documento. Hasta ahora, esto lo haríamos con `position`:

```
.con-position {
  position: relative;
  top: 25px;
  left: 100px;
}
```

Ahora, con `transform` es más fácil y rápido.

```
.con-transform {
  transform: translate(100px, 25px);
}
```

Puedes ver el ejemplo completo en codepen.

10.2.1. La propiedad *translate*

Desde mediados de 2022, las versiones de los navegadores ya soportan la propiedad individual `translate` y no hace falta utilizarla como función dentro de la propiedad `transform`. Con esta propiedad podemos añadir tres valores:

```
translate: 50px;            /* Equivale a translateX(50px) */
translate: 50px 150px;      /* Equivale a translate(50px, 150px) */
translate: 0 150px;         /* Equivale a translateY(150px) */
translate: 50px 20px 10px;  /* Equivale a translate(50px, 20px, 10px) */
translate: 0px 0px 30px;    /* Equivale a translateZ(30px) */
```

10.3. Funciones de rotación

Entre las funciones de rotación, podemos encontrar:

Función	Descripción
`rotateX(x)`	Establece una rotación 2D de 'x' grados sólo en el eje horizontal.
`rotateY(y)`	Establece una rotación 2D de 'y' grados sólo en el eje vertical.
`rotateZ(z)`	Establece una rotación 2D de 'x' grados sobre si mismo.
`rotate(z)`	Alias de la anterior.
`rotate3d(x, y, z, a)`	Realiza una rotación tridimensional.

La mejor forma de entender el comportamiento de estas funciones es utilizar el ejemplo de MDN Web Docs y modificarlo para ver no solo `rotate3d`, también el resto de funciones de rotación.

Observa el siguiente ejemplo de rotación en codepen.

10.3.1. La propiedad *rotate*

Al igual que con la propiedad `translate` que vimos en el apartado anterior, desde mediados de 2022, las versiones de los navegadores ya soportan la propiedad individual `rotate` y no hace falta utilizarla como función dentro de la propiedad `transform`.

Se puede indicar 1 parámetro, 2 parámetros o 4 parámetros, dependiendo de la modalidad a utilizar. Veamos algunos ejemplos:

```
rotate: 45deg;       /* Equivale a transform: rotateZ(45deg);  */

rotate: x 45deg;     /* Equivale a transform: rotateX(45deg);  */
rotate: y 120deg;    /* Equivale a transform: rotateY(120deg); */

rotate: 0 0 1 45deg; /* Equivale a transform: rotateZ(45deg);  */
rotate: 1 0 0 15deg; /* Equivale a transform: rotateX(15deg);  */
rotate: 0 1 1 5deg;  /* Equivale a transform: rotateY(5deg) rotateZ(5deg);  */
```

10.4. Funciones de escalado

Entre las funciones de escalado, podemos encontrar:

Función	Descripción
`scaleX(x)`	Reescala el elemento 'x' veces sólo en horizontal.
`scaleY(y)`	Reescala el elemento 'y' veces sólo en vertical.
`scale(x, y)`	Atajo de las dos anteriores (escalado simétrico).
`scale(x)`	Equivalente al anterior: `scale(x, x)`.
`scaleZ(z)`	Reescala en 3D.
`scale3D(x, y, z)`	Reescala en 3D.

La mejor forma de entender el comportamiento de estas funciones es utilizar el ejemplo de MDN Web Docs y modificarlo para ver no solo `scale3d`, también el resto de funciones de rotación.

También es posible utilizar porcentajes en lugar de números. Por ejemplo:

```
scale: 50% 100%
```

Es sería equivalente a:

```
scale: 0.5 1
```

Además, con la función de escalado de CSS se puede hacer un *efecto espejo* y darle la vuelta a una imagen, por ejemplo, de forma muy sencilla. Basta con utilizar la función `scale(-1)` con valor negativo. Si '1' representa a la imagen tal cual está, '-1' es la imagen invertida.

Puedes utilizar esta función junto con una animación para hacer latir un corazón como en este codepen.

10.4.1. La propiedad `scale`

Al igual que con las propiedades `translate` y `rotate` que vimos en apartados anteriores, desde mediados de 2022, las versiones de los navegadores ya soportan la propiedad individual `scale` y no hace falta utilizarla como función dentro de la propiedad `transform`.

Se puede indicar 1 parámetro, 2 parámetros o 3 parámetros, dependiendo de la modalidad a utilizar. Veamos algunos ejemplos:

```
.element {
  scale: 2;      /* Equivale a transform: scaleX(2);      */
  scale: 2 3;    /* Equivale a transform: scaleY(2, 3);   */
  scale: 2 3 4;  /* Equivale a transform: scale(2, 3, 4); */
}
```

10.5. Funciones de deformación

Entre las funciones de deformación, podemos encontrar:

Función	Descripción
`skewX(xdeg)`	Establece un ángulo de 'xdeg' para una deformación 2D respecto al eje X.
`skewY(ydeg)`	Establece un ángulo de 'ydeg' para una deformación 2D respecto al eje Y.

La mejor forma de entender el comportamiento de estas funciones es utilizar el ejemplo de MDN Web Docs.

Tienes más ejemplos de esta función en w3schools.

10.6. Webgrafía

- https://www.w3schools.com/css/css3_2dtransforms.asp
- https://www.w3schools.com/css/css3_3dtransforms.asp
- https://developer.mozilla.org/es/docs/Web/CSS/CSS_transforms/Using_CSS_transforms
- https://lenguajecss.com/css/transformaciones/transform/

11. Otros

11.1. Reset CSS

Los navegadores tienen sus propias hojas de estilos con el objetivo de hacer las páginas sin estilos más atractivas visualmente.

Estos estilos por defecto no tienen por qué coincidir de un navegador a otro, por lo que si queremos consistencia a través de los distintos navegadores se suelen usar las llamadas hojas de reseteo CSS. Estas hojas sirven para eliminar ciertas características que incorporan por defecto los navegadores. Son ampliamente usadas por frameworks CSS (como Bootstrap) que buscan que todas las páginas construidas con ellos se vean siempre igual independientemente del navegador.

El trabajo posterior tras usar este tipo de hojas es mayor, pero el resultado es más consistente.

Hay muchas hojas de reseteo ya listas para su uso. Dos de las más usadas son:

- Hoja de Reseteo de ERIC Meyer.
- Hoja de Reseteo de HTML5 Doctor.

11.2. Prefijos de navegadores

CSS3 es una especificación modular, se va actualizando por partes. Los navegadores van dando soporte a propiedades experimentales cada uno a su ritmo y lo que hacen es añadir un prefijo a esas propiedades para indicar que ellos sí les han dado soporte antes de que sean soportadas por los demás o antes de que sean parte del estándar. Estos son los llamados **prefijos de navegadores**.

Por ejemplo, CSS3 establece la propiedad `transition`. Es común ver algo así:

```
a {
  -webkit-transition: -webkit-transform 1s;
  transition: -ms-transform 1s;
  transition: transform 1s;
}
```

Hay que tener en cuenta algo importante. Aunque una regla CSS sea soportada por todos los navegadores, no significa que sea soportada por todas las versiones de todos los navegadores. Y como no todo el mundo tiene la versión más actualizada del navegador que está usando, es preferible usar un prefijo a no usarlo.

Los valores más comunes de esos prefijos son:

- `-webkit` para Chrome, Safari, nuevas versiones de Opera y Firefox para iOS.
- `-moz` para navegadores Firefox que no sean para iOS.
- `-o` para versiones antiguas de Opera.
- `-ms` para Internet Explorer y Microsoft Edge.

El problema que surge es que los desarrolladores no sabemos cuándo usar estos prefijos. CSS3 es una especificación larga con muchos módulos y es prácticamente imposible saber cuándo hay que añadir estos prefijos o cuándo las propiedades han dejado de ser experimentales.

Existen varias herramientas que nos facilitan la vida:

- shouldiprefix.com.
- Autoprefixer. Funcional como plugin de VSCode.