INDICE DE REGLAS Y PROPIEDADES CSS

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 @ (como por ejemplo **@media o @import**) y dependiendo de la regla en cuestión, puede tener una sintaxis u otra:

Algunos ejemplos con @media o @import:

```
/* Regla @import para importar estilos CSS de otro fichero */
@import url("index.css");

/* Media query para definir estilos de móvil */
@media screen and (max-width: 640px) {
  .page {
    width: 100%;
  }
}
```

En general, se pueden colocar en cualquier parte de la hoja de estilos, tanto al principio, al medio como al final. Sin embargo, algunas reglas tienen ciertas restricciones. Por ejemplo, como comentaremos, en el caso de la regla **@import debe colocarse en las primeras líneas del fichero**.

Reglas CSS

A continuación, podemos encontrar las siguientes reglas de CSS. Cada una pertenece a temáticas diferentes, en cada enlace profundizamos un poco sobre ellas:

Regla	Descripción	Enlace
`@import`	Incluye una hoja de estilos externa, indicando su URL.	Ver la regla @import
`@media`	Aplica estilos dependiendo de características de la pantalla (responsive).	Ver Media Queries
`@container`	Aplica estilos dependiendo de un elemento contenedor.	Ver Container Queries
`@page`	Modifica propiedades relacionadas con una página impresa.	Ver Medios paginados
`@keyframes`	Crea los fotogramas clave de una animación.	Ver regla @keyframes
`@scope`	Aplica estilos con un límite de ámbito en elementos descendientes.	Ver regla @scope
`@layer`	Establece que estilos se incluirán en una capa (que se fusionará más tarde).	Ver la regla @layer
`@property`	Indica el tipo de dato de una variable CSS, aplicando ciertas ventajas.	Ver regla @property
`@font-face`	Indica una tipografía externa que el navegador debe descargar.	Ver tipografías externas
`@font-feature-values`	Activa/desactiva características especiales en una tipografía	Ver caract. personalizadas
`@font-palette-values`	Permite personalizar los valores por defecto de la propiedad `font-palette`.	-
`@supports`	Establece un bloque de estilos que se aplicará si se cumple la condición.	Ver la regla @supports
`@when` / `@else`	Establece unos estilos si se cumple la condición y otros si no se cumple.	⚠ Aún sin soporte.
`@function`	Crea una función CSS que dada una entrada, produce una salida de CSS.	⚠ Aún sin soporte.
`@mixins` / `@apply`	Crea un grupo de propiedades CSS que se pueden aplicar o reutilizar.	⚠ Aún sin soporte.
`@counter-style`	Indica el estilo que se utilizará en los contadores CSS.	Ver Contadores CSS
`@color-profile`	Define un perfil de color específico que podrá usarse con la función `color()`.	-
`@namespace`	Define un espacio de nombres XML para utilizar en CSS.	-
`@starting-style`	Define valores iniciales de un elemento al que aplicarás una transición CSS.	-

La Regla @Import

La regla @import es una regla de CSS que permite cargar un fichero .css externo, leer sus líneas de código e incorporarlo al archivo actual. **Estas reglas CSS se deben indicar en las primeras líneas del fichero, ya que deben figurar antes de otras reglas CSS.**

Sintaxis de @import

En principio, existen dos sintaxis para cargar ficheros externos mediante la regla @import:

1. Utilizando la función **url()**
2. Indicando simplemente un **string** entre comillas

Formato	Descripción
`@import url("fichero.css")`	Importa el `fichero.css` utilizando la función `url()` de CSS.
`@import "fichero.css"`	Importa el `fichero.css` utilizando un STRING

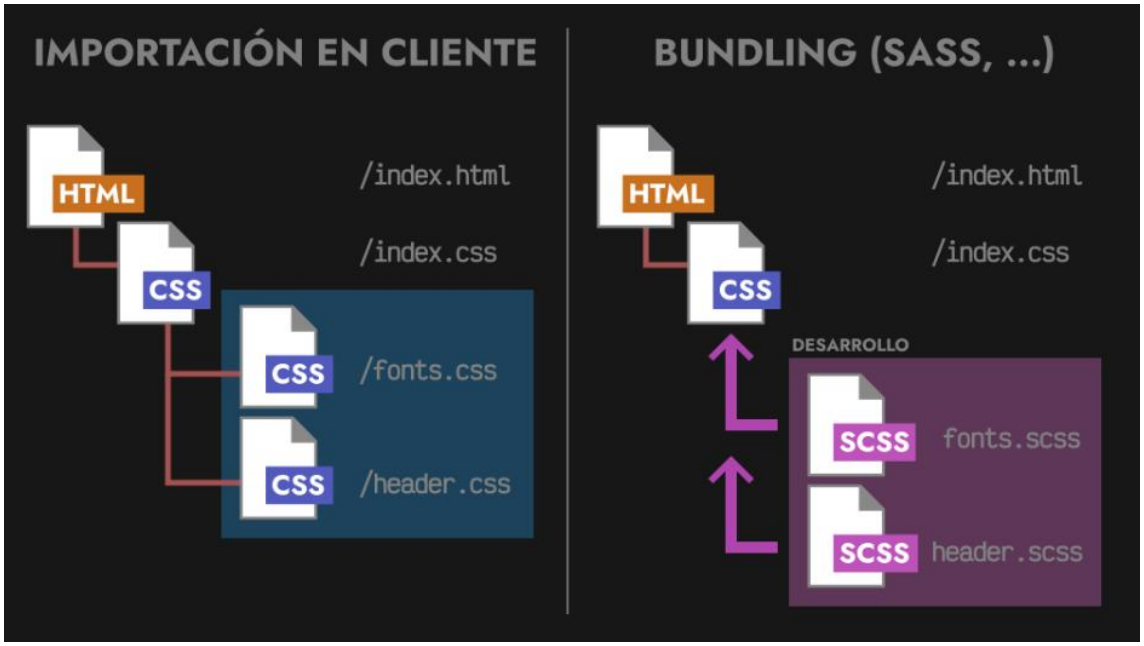
En cualquiera de las dos sintaxis se pueden utilizar tanto los nombres de los archivos, como rutas relativas o absolutas:

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

El soporte de la regla @import es muy bueno, ya que es una regla tradicional en CSS que lleva bastante tiempo. Sin embargo, hay que estar atento a que tiene ciertas modalidades nuevas que podrían tener menor soporte y que veremos a continuación.

Importación en cliente

Un detalle muy importante que es necesario tener siempre en cuenta, es que **la regla @import se evalua en el navegador a la hora de cargar la página. Es decir, cada regla @import equivale a una petición al servidor para descargarse un nuevo archivo .css.**



Tradicionalmente, las importaciones mediante `@import` suponían un problema importante de rendimiento, ya que podría ser mucho mejor incluir todo el código CSS en un solo archivo para reducir el número de peticiones. Sin embargo, actualmente, en páginas que funcionen bajo el protocolo **HTTP/2** o superior, podría no ser un detalle tan crítico.

Por otro lado, herramientas **como Sass, PostCSS o LightningCSS** tienen mecanismos para realizar imports de forma anticipada y generar un sólo fichero con todo el código CSS para que el navegador realice menos peticiones.

Modalidades de @import

Además de su objetivo principal, importar un fichero externo .css, existen algunas modalidades con añadidos interesantes en la regla @import. Dichas modalidades son las siguientes:

Tipo de importación	¿Qué importamos?	Soporte
`@import` URL	Una hoja de estilos CSS externa.	✅ Buen soporte.
`@import` URL `media query`	Una hoja de estilos CSS si coincide con el media query indicado.	✅ Buen soporte.
`@import` URL `supports(condición)`	Una hoja de estilos CSS sólo si el navegador soporta la condición.	⚠️ Cuidado.
`@import` URL `layer(nombre)`	Una hoja de estilos CSS y la colocamos en la capa `nombre`.	✅ Buen soporte.
`@import` URL `layer()`	Una hoja de estilos CSS y la colocamos en una nueva capa anónima.	✅ Buen soporte.

Importaciones con media queries

Tenemos la posibilidad de importar hojas de estilo .css externas a la vez que indicamos posteriormente a la **URL** , una **media query**. Esto nos permitirá que esa hoja de estilos externa se descargue y procese, sólo si estamos en un navegador de un dispositivo que cumple las condiciones de la media query.

Observa algunos ejemplos de un @import con media queries:

```
@import url("mobile.css") (width ⩽ 640px);
@import url("desktop.css") (width ⩾ 1280px);
@import url("print.css") print;
```

1 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.

2 En el segundo caso, hacemos lo mismo con el archivo desktop.css, pero **sólo si el dispositivo tiene como mínimo 1280px de ancho**, presumiblemente un dispositivo de escritorio.

3 En el tercer 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.

Esto es un mecanismo que puede ser bastante interesante para reducir la cantidad de estilos CSS que se descargarán y procesarán en el navegador.

Importaciones condicionales

- **Existe otra forma de importar condicionalmente código CSS, y es utilizando supports tras la de nuestra regla @import**.
- Esto permite que descarguemos y procesemos dicho archivo .css sólo si se cumple la condición del supports, basada en la regla @supports:

```
@import url("flex-fallback.css") supports(not (display: grid));
@supports (display: grid) {
  /* ... */
}
```

En este caso, **si estamos en un navegador antiguo que no soporta grid, descargará y procesará el fichero flex-fallback.css donde colocaremos los estilos alternativos** (utilizando flex, por ejemplo).

En el caso de que lo soporte, hará lo que tenemos en la regla @supports, que aunque se indica en este ejemplo, no tiene relación directa con el @import y **no es obligatorio utilizarlo en conjunto.**

Importaciones en capas

Una característica moderna que han implementado los navegadores, es la posibilidad de utilizar capas en CSS.

Se trata de un sistema similar a las capas de un programa de diseño gráfico (como Photoshop), pero **orientado a código CSS de cara a la Especificidad CSS**.

```
@import url("colaviento.css") layer(framework);

/* Mi código CSS (está en otra capa, separada de framework) */
```

La idea es que **puedes importar un archivo .css y meter su contenido en una capa virtual de CSS**, donde se revisará la especificidad antes de pasar a otra capa.

De esta forma puedes mantener aislados los estilos de un framework con los tuyos propios y no necesitar utilizar !important o reescribir los selectores para forzarlos.

Importaciones desde Javascript

Aunque aún es muy pronto para utilizarlas, **Javascript utiliza un sistema muy similar para importar módulos ESM con contenido Javascript**. Sin embargo, ese estándar se está ampliando para poder importar módulos JSON, CSS o HTML.

La sintaxis sería algo similar a lo siguiente:

```
// Importamos los estilos en un objeto CSSStyleSheet
import styles from "./index.css" with { type: "css" };

// Añade los estilos al documento
document.adoptedStyleSheets.push(styles);
```

Esto nos permitirá en el futuro, trabajar con CSS desde Javascript a nivel nativo, e incluso que sea mucho más sencillo añadir o modificar estilos parciales en nuestra web.

La Regla @Media

Una vez nos adentramos en el mundo del **Responsive Design,** nos damos cuenta en que hay situaciones en las que determinados aspectos o componentes visuales deben aparecer con ciertas diferencias dependiendo del dispositivo donde se están visualizando, ya que no todos los dispositivos tienen los mismos tamaños o características.

Por ejemplo, una zona donde se encuentra el **buscador de la página** puede estar colocada en un sitio concreto en la versión de escritorio, pero en móvil quizás nos interese que ocupe otra zona (*o que tenga otro tamaño o forma*) para aprovechar mejor el espacio de la versión del dispositivo móvil.

¿Qué son las Media Queries?

Existe un concepto denominado media queries, mediante el cual podemos hacer excepciones para que unos determinados estilos de diseño sólo se apliquen si se cumplen una serie de condiciones, generalmente relacionadas con el dispositivo o navegador mediante el cual se está viendo la página.

A fin de cuentas, se trata de una especie de condicionales de diseño, en las que si se cumple la condición se aplicarán unos estilos, y en caso contrario, se aplicarán otros.

La regla @media

Las reglas media queries (abreviadas como MQ) se indican en el código mediante la regla **@media**, indicando la condición en cuestión entre paréntesis. La sintaxis sería la siguiente:

Regla	Descripción
@media (<condición>)	Si se cumple la condición, se aplican los estilos de su interior.
@media not (<condición>)	Si no se cumple la condición, se aplican los estilos de su interior.
@media only (<condición>)	Si se cumple la condición y es un navegador moderno, se aplican los estilos.
@media (<condición>) and (<condición>)	Se se cumplen ambas condiciones, se aplican los estilos.

Observa que, de utilizar **la palabra clave not antes de los paréntesis, invertimos la condición**. Así pues, veamos un pequeño ejemplo donde escribimos las dos opciones anteriores, pero sin entrar en detalles aún en la condición:

```
@media (*condición*) {
  .container {
    background: green;
  }
}

@media not (*condición*) {
  .container {
    background: red;
  }
}
```

En el ejemplo anterior, si se cumple la condición establecida, se aplicará un color verde. Sin embargo, si no se cumple, se aplicará un color rojo. **Recuerda que es similar al funcionamiento de un if / else en programación.**

No olvides que al escribir una regla @media podrías estar sobreeescribiendo los estilos CSS en otro fragmento posterior.

Una buena forma de empezar a escribir MQ sería escribir las reglas @media siempre al final, como excepciones al código anterior.

El número de bloques de reglas `@media` a utilizar depende del desarrollador web, ya que no hay una obligación o norma de utilizar un número concreto. Se pueden utilizar desde un sólo media query, hasta múltiples de ellos a lo largo de todo el documento CSS.

Si lo deseas, es posible establecer múltiples condiciones en las reglas @media. De esta forma, se pueden conseguir situaciones mucho más específicas y flexibles. Ten mucho cuidado si aplicas el not en las condiciones, no sea que niegues de forma incorrecta los casos deseados:

```
@media (*condición*) and (*condición) {
  .container {
    background: orangered;
  }
}
```

Al igual que `not`, también existe la palabra clave `only`. Suele usarse a modo de **hack**. El comportamiento por defecto ya incluye los dispositivos que encajan con la condición, así que con `only` conseguimos que navegadores antiguos (*no la entienden*) no procesen la información de esta regla, consiguiendo una forma de dar estilo sólo en navegadores modernos.

Condiciones de Media Queries

En los ejemplos anteriores hemos indicado ***condiciones*** en el interior de los paréntesis, pero no hemos visto como definir ninguna. Vamos a profundizar en esto.

Aunque existen otras formas, **hoy en día la forma preferida de escribir Media Queries es utilizando la modalidad de rangos de condiciones** (Media Query Range Syntax), mucho más versátiles que las sintaxis anteriores, y mucho menos tediosas.

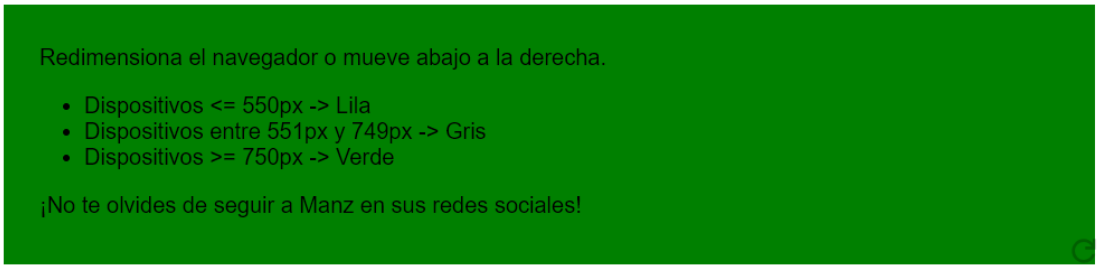
Para ello, vamos a escribir las condiciones utilizando operadores de comparación como <, <=, > o >=:

```
.container {
  background: var(--color, grey);
  min-height: 200px;
  padding: 1rem 2rem;
}

@media (width ⩽ 550px) {
  .container {
    --color: indigo;
  }
}

@media (width ⩾ 750px) {
  .container {
    --color: green;
  }
}
```

```html
<div class="container">
  <p>Redimensiona el navegador o mueve abajo a la derecha.</p>
  <ul>
    <li>Dispositivos ⩽ 550px → Lila</li>
    <li>Dispositivos entre 551px y 749px → Gris</li>
    <li>Dispositivos ⩾ 750px → Verde</li>
  </ul>
  <p>¡No te olvides de seguir a Manz en sus redes sociales!</p>
</div>
```



Observa que en el caso de que la ventana del navegador del dispositivo sea igual o más pequeña que 550px, se verá el fondo de color lila, y en el caso de que sea mayor o igual que 750px se verá verde.

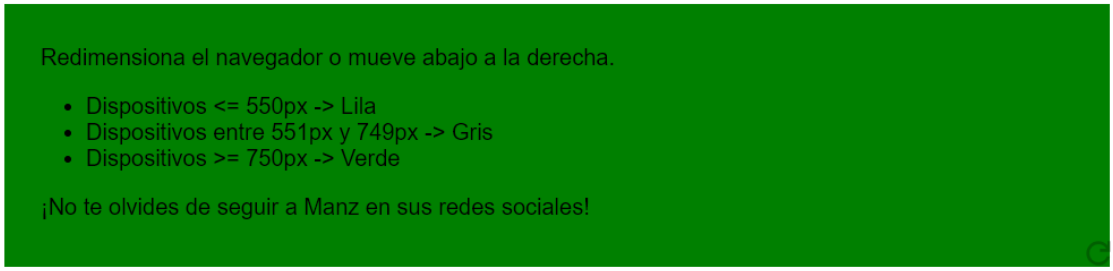
Si la ventana tuviera entre 551px y 749px, no se aplicaría ninguna media query (no cumplen las condiciones) y se quedaría de color gris, que es el valor que tiene indicado por defecto.

Ahora, hagamos una pequeña variación del CSS anterior, y aprovechemos el **CSS Nesting nativo y la anidación de reglas @media:**

```css
.container {
  background: var(--color, grey);
  min-height: 200px;
  padding: 1rem 2rem;

  @media (width ⩽ 550px) { --color: indigo; }
  @media (width ⩾ 750px) { --color: green; }
}
```

```html
<div class="container">
  <p>Redimensiona el navegador o mueve abajo a la derecha.</p>
  <ul>
    <li>Dispositivos ⩽ 550px → Lila</li>
    <li>Dispositivos entre 551px y 749px → Gris</li>
    <li>Dispositivos ⩾ 750px → Verde</li>
  </ul>
  <p>¡No te olvides de seguir a Manz en sus redes sociales!</p>
</div>
```



Como puedes ver, exactamente equivalente al código anterior, pero más compacto y legible. Ten en cuenta que con la sintaxis de rangos también puedes hacer condiciones múltiples de este estilo, en el que el código CSS se aplica sólo si el navegador tiene un ancho de pantalla entre 400px y 800px:

```css
@media (400px ⩽ width ⩽ 800px) {
  /* Código CSS */
}
```

Esta serie de características de sintaxis junto a poder utilizar los operadores >, >=, <, <= en las condiciones de los media queries, hace mucho más intuitivo el escribir MQ CSS. El soporte en la actualidad de esta característica es muy bueno, por lo que no deberías tener problemas.

Media Queries (legacy syntax)

Lo que veremos a continuación es un ejemplo clásico de media queries en el que se utiliza una sintaxis antigua.

Hoy en día se recomienda utilizar la Media Query Range Syntax que vimos antes, pero es posible que te encuentres con una sintaxis similar a la siguiente:

```css
.menu {
  height: 100px;
}

@media (max-width: 400px) {
  .menu {
    background: blue;
  }
}

@media (min-width: 400px) and (max-width: 800px) {
  .menu {
    background: red;
  }
}

@media (min-width: 800px) {
  .menu {
    background: green;
  }
}
```

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

El ejemplo anterior muestra un elemento (con clase menu) con un color de fondo concreto, dependiendo del tipo de medio con el que se visualice la página. Los resultados podrían ser los siguientes (los valores son sólo ejemplos, habría que adaptarlos al caso deseado):

- Azul para resoluciones menores a 400 píxeles de ancho (móviles).
- Rojo para resoluciones entre 400 píxeles y 800 píxeles de ancho (tablets).
- Verde para resoluciones mayores a 800 píxeles (desktop).

Tipos de medios

En algunas ocasiones, queremos indicar que las reglas @media sólo se pongan en funcionamiento en determinados tipos de dispositivos. Son los llamados tipos de medios, que pueden utilizarse en las condiciones de los media queries. Existen los siguientes:

Tipo de medio	Significado
all	Todos los dispositivos o medios. El que se utiliza por defecto.
screen	Monitores o pantallas de ordenador. Es el más común.
print	Documentos de medios impresos o pantallas de previsualización de impresión.
speech	Lectores de texto para invidentes (Antes aural, el cuál ya está obsoleto).

Estos tipos de medios se pueden indicar como una condición más, de modo que podría quedar de la siguiente forma:

```
/* Pantallas menores de 600px */
@media screen and (width ⩽ 600px) {
  /* ... */
}

/* Previsualizaciones y formatos de impresión */
@media print {
  /* ... */
}
```

Quizás encuentres referencias a medios como `braille`, `embossed`, `handheld`, `projection`, `tty` o `tv`. Aunque aún pueden servir, están marcados como obsoletos a favor de los de la lista anterior.

Viewport (Región visible)

Cuando hablamos de Responsive Design muchas veces haremos referencia **al viewport (región visible del navegador**).

Recordemos que con el siguiente fragmento de código HTML estamos indicando que hay que preparar el navegador para el Responsive y que el nuevo ancho de la pantalla será el ancho del dispositivo, por lo que el aspecto del viewport se va a adaptar consecuentemente:

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

Con esto conseguiremos preparar nuestra web para dispositivos móviles y prepararnos para la introducción de reglas media query en el documento CSS. Es importante no olvidar este paso.

Media Queries desde HTML

Por último, hay que tener en cuenta que **los media queries también es posible indicarlos desde HTML, utilizando la etiqueta <link> y el atributo media para establecer la condición**:

```
<link rel="stylesheet"
      href="mobile.css"
      media="(max-width: 640px)">

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

<link rel="stylesheet"
      href="desktop.css"
      media="(min-width: 1280px)">
```

Observa, sin embargo, que, en este caso, el código CSS de las diferentes condiciones queda en un archivo .css diferente, sin embargo, el navegador al cargar la página los descargará todos y los aplicará cuando sea necesario, al igual que lo hace en los ficheros .css.

La Regla @Container

Los media queries o regla @media son un mecanismo de CSS para dar estilo a elementos dependiendo de si se cumple una cierta condición, que generalmente tiene que ver con 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.

¿Qué son los Container Queries?

Los CSS Container Queries son el mismo concepto de las Media Queries, pero en lugar de estar orientados a modificar los estilos dependiendo del tamaño de la página o dispositivo, lo hace 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.

El contenedor

- Para empezar, **tenemos que determinar cuál será el elemento contenedor al que vamos a hacer referencia.**
- En dicho elemento, **necesitaremos establecer las siguientes propiedades, donde container-name es siempre obligatoria**:

Propiedad	Valores	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 linea...).

La propiedad container-name le da un nombre de contenedor al elemento en el que nos encontramos. **Sin este nombre no podremos hacer referencia luego, en la regla @container**.

Por otro lado, **la propiedad container-type, si la definimos, establecerá el tipo de tamaño que va a tener el contenedor en cuestión**, **donde puede ser size para elementos de tipo bloque, o inline-size para elementos de tipo en línea**.

Por ejemplo, **en el caso de establecer un elemento inline-size, no tendrá en cuenta la altura**, si la colocamos en la regla @container, al contrario que si usamos size.

```
.container {
  container-name: parent;
  container-type: inline-size;
}
```

Ver propiedad display

Atajo: La propiedad container

Una forma de escribir menos código es utilizar la propiedad de atajo container, a la cuál le podemos indicar **dos valores**:

- el valor de la propiedad **container-name y el valor de la propiedad container-type**, **siempre separadas por un /**:

Propiedad	Valor
container	container-name / container-type

Muchas veces no necesitaremos establecer el valor de la propiedad container-type, por lo que podemos indicarle simplemente el primer valor, de la propiedad container-name y nada más.

La regla @container

Una vez tenemos nuestro contenedor definido, debemos establecer una regla @container que, de forma muy similar a las reglas @media va a establecer una condición dependiendo de un elemento contenedor específico.

Regla	Descripción
@container <nombre del contenedor> (<condición>)	Establece una condición para aplicar estilos al contenedor concreto.

Por ejemplo, en este ejemplo que tenemos a continuación, vamos a establecer nuestro elemento con clase .container con un elemento contenedor parent, y creamos una regla @container que afecta a los hijos del contenedor parent cuando, como máximo, tenga 500px de ancho:

```
.container {
  container: parent;
}

@container parent (max-width: 550px) {
  .item {
    background: blue;
  }
}
```

Vamos ahora a aplicar este detalle a un ejemplo completo.

Observa que la tarjeta aparece en formato horizontal. Sin embargo, si forzamos a reducir el ancho de la ventana, comprobaremos que el elemento reacciona al cambio y se muestra en versión vertical. Esto puede parecer lo mismo que un media query, pero realmente podemos hacer lo mismo reduciendo el tamaño de la tarjeta (esquina inferior-derecha), y también reacciona a su cambio de tamaño.

Esto ocurre porque hemos usado un Container Query en lugar de un Media Query:

```
body {
  background: steelblue;
  height: 625px;
}

.container {
  container: card / inline-size;
  max-width: 1024px;
  min-width: 250px;
  overflow: hidden;
  resize: horizontal;
  border: 1px solid white;
  box-shadow: 4px 4px 10px #0005;
  background: white;
  color: black;

  & .item {
    display: flex;
    justify-content: center;
    align-items: center;
    background:
      linear-gradient(to right, transparent 180px, white 180px),
      linear-gradient(to bottom, #9146eb 50%, #ea17cf 50%);

    @container card (max-width: 550px) {
      flex-direction: column;
      background: linear-gradient(#9146eb, #ea17cf 180px, transparent 180px);
    }

    & img {
      position: relative;
      width: 180px;
      height: 180px;
    }

    & .data {
      margin: 1em;
    }
```

```html
<p>Move from bottom-right corner</p>

<div class="container">
  <div class="item">
    <img src="https://manz.dev/apple-touch-icon.png" alt="ManzDev">
    <div class="data">
      <h1>Manz.dev</h1>
      <p>Página del canal de Twitch de ManzDev, donde se habla de Desarrollo web, front-end y
    </div>
  </div>
</div>
```

En este caso, cuando el elemento .item se encuentre dentro del contenedor .container y este último, tenga menos de 500px de ancho, el estilo del elemento .item cambiará a lo definido en la regla @container

Unidades de contenedores

Cuando nos encontramos en el interior de una regla @container podemos utilizar ciertas unidades especiales como **cqw, cqh, cqi, cqb, cqmin o cqmax**.

Básicamente, la idea es que, si desconocemos el tamaño concreto del contenedor, podemos utilizar estas medidas para aplicar un porcentaje de su tamaño.

De esta forma, si utilizamos 50cqw, significa que va a establecer un tamaño del 50% del ancho del contenedor.

Funcionan de forma muy similar a las unidades vw, vh, vmin y vmax.

Unidad	Descripción
`cqw`	**Container Query Width**. Porcentaje relativo al ancho del contenedor.
`cqh`	**Container Query Height**. Porcentaje relativo al alto del contenedor.
`cqi`	**Container Query Inline Size**. Porcentaje relativo al tamaño en línea.
`cqb`	**Container Query Block Size**. Porcentaje relativo al tamaño de bloque.
`cqmin`	**Container Query Mínimo**. Valor más pequeño entre `cqi` y `cqb`.
`cqmax`	**Container Query Máximo**. Valor más alto entre `cqi` y `cqb`.

Las unidades cqi y cqb son las propiedades lógicas equivalentes al ancho y alto. PROPIEDADADES LÓGICAS CSS (VER).

Condiciones con función style()

Aunque aún no tiene muy buen soporte, **es posible utilizar la función css style() para realizar comprobaciones concretas en variables CSS y determinar si se deberían aplicar los estilos.** Por ejemplo, observa el siguiente ejemplo:

```css
.container {
  container: parent;
}

@container parent (max-width: 500px) and style(--responsive: true) {
  .item {
    background: blue;
  }
}
```

La función **style(--responsive: true)** comprueba si existe una variable **--responsive** y está establecida a true, **cosa que se puede hacer desde nuestro código CSS, o establecerla desde Javascript y guardarla en una variable CSS**, permitiendo hacer nuestras condiciones de contenedores mucho más flexibles.

Medios paginados

Media queries para adaptar contenido a páginas|

Existe una particularidad de CSS **denominada CSS Paged Media**, que no es más que un conjunto de características y propiedades específicas para medios paginados, como por ejemplo podrían ser, **documentos impresos o la exportación en archivos .pdf**.

La regla @page

Existen varias propiedades CSS interesantes para estos medios, que pueden ser utilizadas dentro de un bloque @page:

Propiedad	Valores	Significado
size	auto | tipo documento | SIZE SIZE	Indica el ancho y alto de cada página.
margin	SIZE	Idéntica a la propiedad margin que conocemos.

Mediante la propiedad **size** se puede establecer el formato de la página, tanto mediante los **valores predefinidos portrait o landscape (vertical o apaisado), como especificando las medidas mediante unidades:**

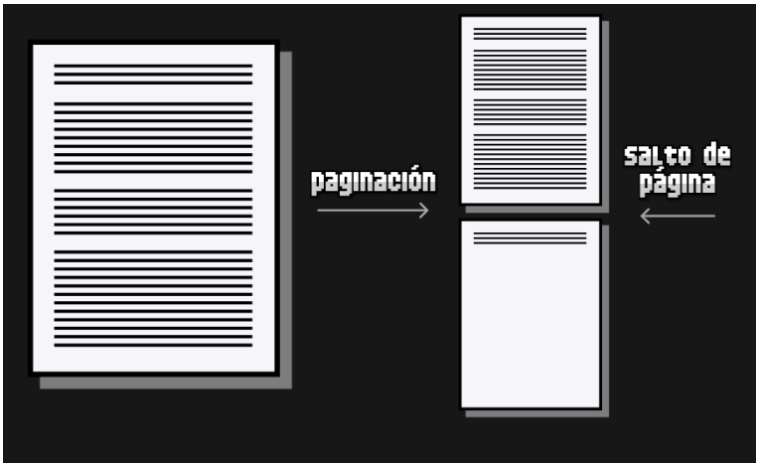
```
@page {
  size: 8.27in 11.69in;  /* DIN A4 */
  margin: 0.5in 0.5in 0.5in 0.5in;
}
```

De forma adicional, también puedes indicar el tipo de documento mediante algunas palabras clave como valor de la propiedad **size (las cuales se pueden usar junto a landscape o portrait para definir su orientación):**

Valor	Medidas	Significado
A5	148mm×210mm	Documento de mitad de tamaño de un folio DIN A4.
B5	176mm×250mm	Documento de tamaño entre A5 y A4.
A4	210mm×297mm	Documento de exactamente el tamaño folio DIN A4.
B4	250mm×353mm	Documento de tamaño entre A4 y A3.
A3	297mm×420mm	Documento del doble de tamaño de un folio DIN A4.
letter	215.9mm×279.4mm	Documento de tamaño de las cartas americanas.
legal	215.9mm×355.6mm	Documento de tamaño de notificación legal americana.
ledger	279.4mm×431.8mm	Documento de tamaño libro americano.

Nota: Cualquier propiedad utilizada dentro de una regla @page que no sea una de las anteriores será ignorada.

Una vez tenemos definida la estructura de cada página del documento, pasaremos de una página web que tiene una estructura continua a un documento paginado, donde cada cierto tamaño tiene un salto de página que interrumpe el contenido.



Al establecer dichos saltos de página, el contenido se adapta al tamaño de cada página especificado, **pero puede ocurrir que, debido a los límites impuestos por los saltos de página, un cierto contenido se corte a la mitad** y se vea continuado en la siguiente página, comportamiento no deseable en prácticamente el 100% de los casos.

Orphans & Widows

Existen casos en los que el comportamiento descrito anteriormente no sea un problema, sin embargo, pueden darse muchos casos en los que sí lo sería:

- Una imagen o una tabla empieza al final de una página (se muestra cortada).
- Un párrafo de texto comienza en la última línea de una página (es poco atractivo).
- Un nuevo tema comienza a mitad de página (lo ideal sería comenzar a principio de página).

Para evitar estos problemas de paginación se suelen utilizar algunas propiedades interesantes:

Propiedad	Valores	Significado
orphans	NUMBER	Número de líneas «huérfanas» permitidas (antes de salto)
widows	NUMBER	Número de líneas «viudas» permitidas (después de salto)

Mediante **la propiedad orphans (líneas huérfanas)** indicamos hasta que número de líneas no vamos a permitir que aparezcan de forma aislada al final de una página.

Por ejemplo, si especificamos **orphans: 3, significa que cualquier párrafo cortado con el salto de página, debe mostrar más de 3 líneas al final de la página**. En caso contrario, se moverá el párrafo completo a la siguiente página.

```
p {
  orphans: 3;
  widows: 3;
}
```

Por otra parte, la propiedad widows (líneas viudas) es la propiedad opuesta a orphans. Si al principio de una página quedan menos de ese número de líneas indicado en widows, se ajusta para que lo supere.

Saltos de página

Por último, mencionar las propiedades de salto de página, muy útiles para evitar comportamientos que pueden quedar poco atractivos en un documento paginado.

Estas propiedades pueden utilizarse tanto en medios paginados, como en multicolumnas, donde tiene un soporte menor:

Propiedad	Valores	Significado
break-inside	auto | avoid	Evita cortar elementos en un salto de página
break-before	auto | page | avoid | left | right	Evita o crea un salto de página antes del elemento
break-after	auto | page | avoid | left | right	Evita o crea un salto de página después del elemento

La propiedad break-inside **se utiliza para evitar que un cierto elemento pueda ser cortado a la mitad en un salto de página,** como por ejemplo imágenes, tablas o fragmentos de código:

```
@media print {
  table,
  img,
  pre,
  code {
    break-inside: avoid;
  }
}
```

De forma similar, las propiedades break-before y break-after **se utilizan para evitar o forzar al generar el documento paginado que ciertos elementos se mantengan antes o después del salto de página.**

Fragmentos rotos

Se considera un fragmento roto, cuando tenemos un elemento que, por las medidas del padre, se «interrumpe» y continua en la línea siguiente, pero por su naturaleza, da la impresión que se ha roto.

Para cambiar este comportamiento, podemos utilizar la propiedad box-decoration-break:

Propiedad	Valores	Significado
box-decoration-break	slice | clone	Determina si se debe «partir» el elemento, o clonar en varias partes.

Un ejemplo claro del uso de esta propiedad, se puede ver en el siguiente ejemplo:

```html
<div class="container">
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Quam excepturi
    similique iure voluptate beatae cum libero, <span>assumenda expedita
    possimus, neque hic minus</span>, enim eveniet temporibus quibusdam
    fuga cupiditate laborum maxime.</p>
</div>

<style>
.container {
  width: 200px;
}

span {
  border: 3px solid black;
  background: gold;
  box-decoration-break: clone;
}
</style>
```

Observa que el elemento , un elemento que tiene una naturaleza en línea, se podría ver «partido» si tiene definido un borde y/o un fondo.

Este es el comportamiento por defecto de la propiedad, que está establecida al valor slice.

Podemos indicarle el valor clone, si lo que queremos es que el navegador cree varios clones independientes y «cierre» las partes, en lugar de mostrarla con aspecto de estar roto o «partido».

Pseudoclases de página

También existen algunas pseudoclases específicas para paginación, **que permiten hacer referencia a páginas concretas del documento generado**, pudiendo aplicar estilos sólo a este tipo de páginas:

Pseudo-clase	Significado
:first	Primera página del documento paginado.
:left	Páginas izquierdas del documento paginado.
:right	Páginas derechas del documento paginado.
:blank	Páginas en blanco (generadas por los saltos de página).

Sólo algunas propiedades CSS como margin, padding, background u otras pueden estar permitidas dentro de estas pseudoclases especiales.

Regla @keyframes

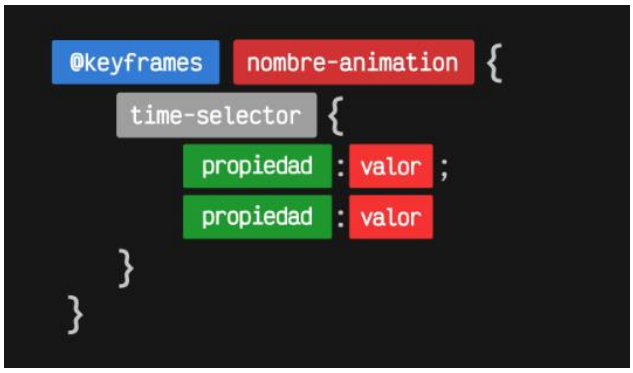
Cuando ya hemos utilizado propiedades CSS como **animation (o de su familia, animation-*),** nos falta una parte muy importante: **definir los fotogramas de la animación**.

Una animación está formada por varios fotogramas, una secuencia de imágenes (30-60 fotogramas por segundo, por ejemplo) **que mostradas una detrás de otra generan el efecto de movimiento que conocemos de una animación**.

En CSS, los fotogramas se crean a partir de propiedades CSS, y no hace falta definir tantos fotogramas. Sólo crearemos fotogramas clave y el resto de fotogramas los generará el navegador.

Regla @keyframes

Para definir esos fotogramas clave, utilizaremos la regla **@keyframes**, la cuál es muy sencilla de utilizar. 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, y ya veremos que pueden definirse muchos en una misma animación.

Selectores from y to

Como vemos, dicha sintaxis tiene dos partes interesantes, ya que las reglas (propiedad, valor) ya las conocemos en CSS.

Las partes principales por las que debemos comenzar son:

Parte	Descripción
@keyframes STRING name	Regla para darle un nombre y definir los fotogramas clave de una animación.
from	Fotograma clave inicial con los estilos CSS a aplicar. Equivalente a 0%.
to	Fotograma clave final con los estilos CSS a aplicar. Equivalente a 100%.
PERCENT	Porcentaje especifíco de la animación con los estilos CSS a aplicar. Permite decimales.

Esto siempre se ve mejor con un ejemplo, así que vamos a mostrarlo:

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

Como vemos, hemos nombrado change-color la animación, que parte de un primer fotograma clave con el fondo rojo hasta un último fotograma clave con fondo verde.

Cuidado con olvidarse de la s final de @keyframes, ya que está en plural. Respecto al nombre de la animación, lo recomendable es utilizar **kebab-case** a la hora de nombrarla y que represente bien lo que hace.

Kebab-case

Kebab Case se utiliza para nombrar palabras separadas por - (guión) y todas las letras minúsculas.

Generalmente, Kebab Case se utiliza para nombrar CSS

Ej : send-mail

Recuerda que no basta con definir la animación mediante @keyframes, también que hay que asociar la animación al elemento o etiqueta HTML al que queremos aplicársela:

```css
.element {
  width: 100px;
  height: 100px;
  animation: change-color 1.5s linear infinite;
}

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

```html
<div class="element"></div>
```

La magia de la regla **@keyframes** es que nosotros solo declaramos los fotogramas clave, mientras que el navegador irá generando los fotogramas intermedios para conseguir una animación fluida.

Truco: Si una animación va a golpes, es muy posible que te hayas pasado con el tiempo de duración de la misma. Intenta utilizar valores bajos como `0.25s`, `0.5s`, `1s` y no valores mucho más altos.

Selectores porcentuales

Los selectores from y to son muy similares a colocar 0% y 100%, así que los modificaremos y de esta forma podremos ir añadiendo nuevos fotogramas intermedios. Vamos a añadir un fotograma intermedio e indentando, ahora sí, correctamente el código:

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

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

```html
<div class="animated"></div>
```

En este caso, la animación va a progresar por tres fotogramas clave, por lo que irá desde el color rojo, al color amarillo, para finalizar en el color verde.

Esta animación funcionará durante 2 segundos y se realizará una sola vez (**por defecto las animaciones solo se realizan una vez, salvo que le indiques infinite o el número deseado**).

Truco: Si tienes fotogramas que van a utilizar los mismos estilos que uno anterior, siempre puedes separarlos con comas, por ejemplo: `0%, 75% { ... }`, que utilizarían dichos estilos al inicio de la animación y al 75% de la misma.

Regla @scope

Históricamente, uno de los principales problemas para un programador respecto a CSS, es el tema de la cascada y la especificidad. Cuando desarrollamos con CSS y estamos creando código CSS a pequeña escala no suelen haber demasiados problemas. Sin embargo, cuando el código CSS crece, todo comienza a ser diferente.

¿Por qué usar @scope?

Antes de explicar cómo funciona **@scope**, convendría entender porque (o para qué) lo necesitamos.

El caso más claro es aquel donde estamos creando estilos CSS en una parte concreta de nuestra página. Todo va maravillosamente bien y funciona correctamente.

Tenemos el siguiente fragmento de HTML:

```html
<div class="parent">
  <div class="element">Element</div>
</div>
```

Nuestra idea es darle estilo al elemento **con clase .parent, y a los elementos con clase .element que estén en su interior.**

En principio pensamos que no vamos a tener más elementos .element en el interior de un elemento con clase .parent, por lo que no vemos mayor problema y continuamos con nuestra tarea.

El problema es que en este punto no solemos tener una visión global de lo que estamos haciendo, y estamos metidos en resolver un problema local que suele resolverse bien.

Establecemos los siguientes estilos CSS:

```css
.parent {
  background: black;
  color: #fff;
}

.parent .element {
  background: indigo;
  padding: 15px;
  color: gold;
}
```

A medida que nuestro código HTML y CSS crece y pasa el tiempo, se necesitan hacer cambios, modificar y añadir nuevos elementos HTML y nuevo código CSS.

La naturaleza de CSS es global, y cualquier cambio que hagamos repercute en todo el documento, salvo que lo tengas bien acotado.

Podría ocurrirnos que, posteriormente, comenzamos a crear otra zona de la página y creamos un nuevo elemento con clase element, que queremos que tenga otro estilo diferente.

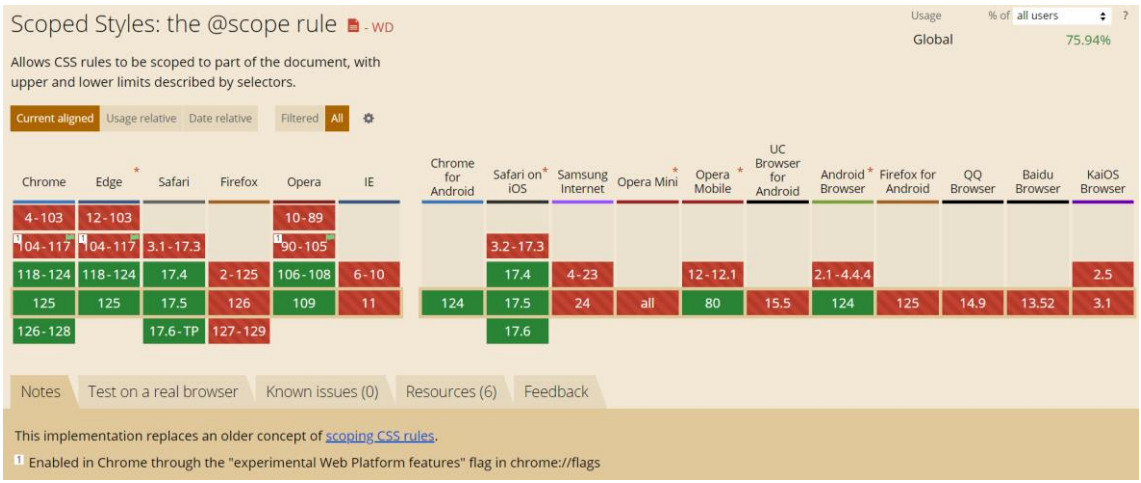
Como efecto secundario, el CSS de cada uno se va a mezclar, porque coinciden sus nombres (es complejo predecir esto).

Para solucionar esto, se han ideado multitud de sistemas:

- Añadir un !important (la más usada, y casi siempre, la peor solución)
- Reorganizar los selectores y hacerlos más complejos (más espeCSSíficos)
- Reescribir o cambiar los nombres de los selectores para que no coincidan.
- Utilizar nomenclaturas o metodologías para evitar coincidencias de clases CSS (BEM, CUBECSS, etc...)
- Usar herramientas/librerías que renombran para evitar colisiones (CSS Modules, CSS-in-JS...)
- Utilizar Shadow DOM para impedir acceso al CSS del elemento desde dentro o fuera.

Sin embargo, todas tienen su complejidad, ventajas y desventajas. Ahora, con la regla @scope tendremos un mecanismo más, pero esta vez muy simple, con CSS nativo, sin recurrir a complejas herramientas, metodologías o tecnologías.

Recuerda que la regla @scope aún está en fase experimental y aún debe ser soportada por otros navegadores:



La regla @scope en línea

Un detalle interesantísimo sobre la regla @scope es que puede trabajar en niveles de profundidad concretos del HTML, dependiendo de donde declares los estilos, mediante una etiqueta <style>.

- Observa el siguiente ejemplo, donde se declara la etiqueta <style> en el interior de un con clase .container.
- La regla @scope en una etiqueta <style> actúa sólo en el ámbito de su etiqueta padre, por lo que estos estilos están limitados sólo a la etiqueta interior:

```
<p>Hola, soy <strong>ManzDev</strong> exterior.</p>

<div class="container">
  <style>
    @scope {
      strong {
        background: orangered;
        color: white;
        padding: 0.2rem;
      }
    }
  </style>
  <p>Hola, soy el <strong>ManzDev</strong> interior.</p>
</div>
```

Hola, soy **ManzDev** exterior.

Hola, soy el **ManzDev** interior.

Repasemos varios detalles clave aquí:

- El primer está fuera de .container.
- El segundo está dentro de .container.
- El @scope se declara en un <style> dentro de .container, por lo que sólo le afecta a él.

Esto es la funcionalidad básica de @scope, sin embargo, no es necesario utilizarlo en línea en una etiqueta <style>, tiene muchas posibilidades más que veremos a continuación.

La regla @scope

- La regla @scope es muy sencilla de utilizar.
- Por defecto, cuando escribimos CSS, el navegador interpreta los estilos para todo el documento.
- **La regla @scope nos da un sistema para delimitar el ámbito en el que afectan los estilos, de modo que podemos marcar un principio y un final de ámbito en nuestro DOM, es decir, en los niveles de profundidad de nuestras etiquetas HTML.**

La estructura de esta regla es la siguiente:

```
@scope (límite inicial) to (límite final) {
  /* Estilos CSS */
}
```

- **La regla @scope define que el CSS que contiene actuará en un ámbito limitado.**
- El límite inicial (opcional) define desde donde se aplicarán los estilos.
- El límite final (opcional) define hasta donde (excluído) se aplicarán los estilos.
- Límite inicial de @scope

Veamos un fragmento de código utilizando la regla @scope. Entre paréntesis delimitaremos el ámbito inicial para el que se van a aplicar los estilos CSS:

```
@scope (.parent) {
  .element { background: orangered; }
}
```

```
<div class="parent">
  <h2 class="element">Hello, ManzDev (inside)</h2>
</div>
<h2 class="element">Hello, ManzDev (outside)</h2>
```

Hello, ManzDev (inside)

Hello, ManzDev (outside)

Este código aplicará los estilos a cualquier elemento con clase .element que esté en el interior de un elemento con clase .parent.

Vamos, prácticamente igual que la siguiente opción 1, o incluso la opción 2, que tiene menor especificidad por el uso del combinador funcional :where():

```css
/* Opción 1 */
.parent .element {
  background: orangered;
}

/* Opción 2: Menos específico */
:where(.parent) .element {
  background: orangered;
}
```

Pero... ¿entonces? Si esto ya se podía hacer hasta ahora... **¿Dónde es verdaderamente útil y novedosa la regla @scope?**

Pues, por ejemplo, cuando delimitamos también el final del ámbito.

Límite final con to

Observa el siguiente caso, donde queremos delimitar el inicio donde **se aplicarán unos estilos y también el final**. Para ello, compliquemos un poco más nuestro HTML anterior.

Observa que ahora tendremos 4 elementos .element:

- El primer .element está dentro de .grandparent pero fuera de los demás.
- El segundo .element está dentro de .grandparent y .parent.
- El tercer .element está dentro de .grandparent, .parent y .child.
- El cuarto .element fuera fuera de todos.

Ahora, vamos a escribir un nuevo fragmento de código CSS con ámbito limitado.

Observa que en este caso indicamos que el ámbito de los estilos va a aplicarse desde el elemento con clase .grandparent hasta el elemento con clase .child (excluído):

```css
@scope (.grandparent) to (.child) {
  .element {
    background: orangered;
    color: white;
    padding: 5px;
  }
}
```

```html
<div class="grandparent">
  <div class="element">Element GrandParent</div>
  <div class="parent">
    <div class="element">Element Parent</div>
    <div class="child">
      <div class="element">Element Child</div>
    </div>
  </div>
</div>
<div class="element">Element outside</div>
```

Si realizas este proceso, se dará estilo al primer y segundo .element, pero no al tercero ni al cuarto. Esto ocurre porque se hace mediante niveles de profundidad de los elementos HTML.

- Al cuarto .element no se le da estilo porque está fuera de .grandparent.
- Al tercer .element no se le da estilo porque está en nivel 4, nosotros pedimos «desde .grandparent» (nivel 1) hasta .child» (nivel 3).

Si reescribieramos el selector de límite como `@scope (.grandparent) to (.child > .element > *)` entonces si estaríamos obteniendo el tercer elemento `.element`, porque estamos seleccionando hasta el contenido del elemento `.element` dentro de `.child`.

Límite final inexistente

Podría darse la circunstancia de que definamos un límite de ámbito final que realmente no existe en el HTML, como por ejemplo el siguiente:

```
@scope (.grandparent) to (.grandson) {
  .element {
    background: orangered;
    color: white;
    padding: 15px;
  }
}
```

Como el elemento con clase .grandson no existe, el navegador interpretará la regla scope como si no tuviera límite final, es decir, como @scope (.grandparent).

De esta forma, estaría seleccionando todos los .element que se encuentren en el interior de un .grandparent.

Múltiples límites

Otro detalle interesante sobre la regla @scope es que podemos utilizar múltiples valores, de forma similar al combinador lógico :is(). Observa el siguiente ejemplo:

```
@scope (.post-container, .comments-container) to (.items) {
  .element {
    background: indigo;
    color: white;
  }
}
```

```html
<div class="post-container">
  <div class="element">Element 1 (Selected)</div>
  <div class="items">
    <div class="element">Element 2 (Not selected)</div>
    <div class="element">Element 3 (Not selected)</div>
  </div>
</div>

<div class="element">Element 1 (Not Selected)</div>

<div class="comments-container">
  <div class="element">Element 1 (Selected)</div>
  <div class="items">
    <div class="element">Element 2 (Not selected)</div>
    <div class="element">Element 3 (Not selected)</div>
  </div>
</div>
```

En este ejemplo, dentro de los selectores de los paréntesis indicamos múltiples valores, por lo que podemos establecer selecciones múltiples y no atarnos a elecciones más sencillas. Esto puede resultar especialmente útil de cara a la mantenibilidad.

La pseudoclase :scope

La pseudoclase :scope se puede utilizar en las reglas @scope para hacer referencia al ámbito donde nos encontramos.

Por ejemplo, observa que en el siguiente ejemplo, el ámbito seleccionado es desde .grandparent hasta .child (excluído), por lo tanto, **al indicar la pseudoclase :scope estamos haciendo referencia al ámbito inicial, es decir, a grandparent**:

```css
@scope (.grandparent) to (.child) {
  :scope > .element {
    background: orangered;
    color: white;
    padding: 15px;
  }
}
```

En este caso, el navegador aplicaría los estilos únicamente al .element del interior de .grandparent.

Sin embargo, si le quitamos el >, observa que aplicaría estilos al .element del .grandparent y del .parent, ya que ambos están dentro de .grandparent. El de .child se obvia, porque recuerda que el límite final está excluído.

Esta característica podría ser realmente útil para simplificar código cuando tenemos múltiples valores en los límites del scope.

Ten en cuenta que también podríamos seleccionar elementos según sus ancestros:

```
@scope (.grandparent) to (.child) {
  .dark :scope > .element {
    background: #222;
    color: white;
    padding: 15px;
  }
}
```

En este caso, seleccionamos los .element que estén dentro de un .grandparent que tenga algún padre con la clase .dark.

Regla @layer

La regla @layer de CSS, permite declarar una capa de cascada.

Estas capas son muy similares y funcionan de forma muy parecida a las capas de cualquier editor gráfico.

Permiten agrupar código CSS en el interior de una capa, y finalmente, fusionarlo todo manteniendo el orden especificado, algo que puede hacer mucho más fácil el organizar CSS, sobre todo de cara a la especificidad.

Veamos un ejemplo de la sintaxis de la regla @layer:

```
@layer reset {
  body {
    margin: 0;
    box-sizing: border-box;
  }
}
```

En este caso, estamos creando una capa reset que va a incluir código CSS que normalmente hace un reseteo en ciertas propiedades sobre la forma que funciona un navegador.

El nombre reset lo establece el desarrollador, y puede ser cualquier otro nombre que desee.

Esto significa que, a partir de ahora, existirá una capa reset que incluye los estilos indicados en el interior de dicha regla.

Pero, aparte de la creación de una capa, ¿cuál es su finalidad? ¿Para qué necesitamos las capas?

Básicamente para poder reordenarlas y evitar problemas de especificidad que pueden aparecer a la larga.

Ten en cuenta que esta característica aún es experimental y puede no estar soportada en algunos navegadores.

Crear capas en CSS

Primero, vamos a conocer las múltiples formas de crear capas en CSS utilizando la regla **@layer**, **o incluso utilizando la regla @import**. Veamos que sintaxis tenemos disponibles:

Formato	Descripción
`@layer`	Crea una capa CSS anónima (`sin nombre`).
`@layer nombre`	Crea una capa CSS con el nombre indicado.
`@layer nombre.subcapa`	Crea una capa CSS con `nombre`, con una subcapa anidada `subcapa`.
`@layer nombre1, nombre2, nombre3...`	Declara y establece un orden para varias capas CSS.
`@import(fichero.css) layer(nombre)`	Importa un archivo CSS y lo introduce en la capa indicada.

Utilizando la regla @layer podemos crear una capa CSS con un nombre determinado:

```css
@layer utils {
  .primary {
    background: #34a;
    border: 2px outset #6381db;
    color: #fff;
    padding: 5px 10px;
    border-radius: 6px;
  }
}
```

Al crear capas, de forma implícita estamos indicando el orden de las capas (orden en el que han sido definidas).

Esto es una parte muy importante de la creación de capas en CSS, ya que determina el orden en el que se van a evaluar. Sin embargo, veremos más adelante que ese orden puede modificarse.

Capas anónimas

Si en lugar de escribir el nombre de la capa (utils en nuestro ejemplo) no escribimos nada, estamos creando una capa anónima sin nombre, que es lo mismo que ocurre si colocamos CSS sin ninguna capa, al final se creará una capa anónima donde se incluirá ese código CSS.

La sintaxis explícita sería la siguiente:

```css
@layer {
  .primary {
    background: #34a;
    border: 2px outset #6381db;
    color: #fff;
    padding: 5px 10px;
    border-radius: 6px;
  }
}
```

Ten en cuenta que en el siguiente apartado veremos que existen formas de hacer referencia a capas para reordenarlas o añadir más contenido en dicha capa.

En el caso de crear **capas anónimas**, no existe ninguna forma de hacer referencia a ellas posteriormente para añadir más código CSS o reordenarlas. Recuerda que si creamos múltiples capas anónimas como en el ejemplo anterior, el navegador creará múltiples capas anónimas diferentes.

Orden de las capas

Podemos cambiar el orden de las capas, si establecemos una regla `@layer` con las diferentes capas separadas por coma.

Hay que asegurarse que esto ocurre antes de la creación de las capas, ya que una vez están declaradas, no se puede cambiar su orden:

```css
@layer reset, texts, theme;

@layer reset {
  button {
    padding: 30px;
  }
}

@layer theme {
  .primary {
    background: #34a;
    border: 2px outset #6381db;
    color: #fff;
    padding: 5px 10px;
    border-radius: 6px;
  }
}

@layer texts {
  .primary {
    color: red;
  }
}
```

```html
<button class="primary">First button</button>
<button class="primary">Second button</button>
```

Si en el caso anterior, no definimos la primera línea, el orden de las capas sería reset, theme, texts, sin embargo, al haberlo definido, el orden de las capas del ejemplo anterior se establece en reset, texts, theme.

Observa que la diferencia aparente es que el texto de los botones aparecen en color rojo.

Para ello, hay que comprender como funciona la especificidad de CSS.

Este ejemplo es muy sencillo, y simplemente, con la primera regla @layer cambiamos el orden en el que se procesan las capas, consiguiendo darle prioridad a la capa theme porque está en último lugar, sobreescribiendo los estilos del color de texto de la capa texts.

En el caso de indicar múltiples veces una misma capa, el navegador fusionará los estilos en la misma capa.

Esto permitirá que en algunos casos podamos añadir más estilos a una capa ya definida.

Imagina que añadimos el siguiente código CSS al ejemplo anterior:

```
@layer theme {
  .primary {
    color: gold;
  }
}
```

Observa que, en este caso, el color de texto del botón será gold.

> Ten en cuenta que cualquier estilo declarado sin capa, independientemente del orden de aparición, se agrupará en una **capa anónima** y se aplicará **siempre al final** del resto de capas declaradas.

La especificidad en capas CSS

Observa el siguiente ejemplo. En él, encontrarás tres grupos de código CSS donde se utiliza el selector.

En estos bloques de código no suele haber dudas, ya que, al ser el mismo selector, se aplica el orden y la herencia, y simplemente se fusionan los estilos, sobrescribiendo el último a los anteriores de coincidir las propiedades:

```
.primary {
  color: red;
}

.primary {
  background: #34a;
  border: 2px outset #6381db;
  color: #fff;
  padding: 5px 10px;
  border-radius: 6px;
}

.primary {
  margin: 20px;
  color: gold;
}
```

```html
<button class="primary">Click me!</button>
```

Sin embargo, modifiquemos las clases para hacer más específicos los selectores.

En cada grupo seleccionamos el mismo elemento, pero con diferente especificidad:

- **Primer grupo**: Botones button que tienen un atributo class y clase primary (especificidad 021)
- **Segundo grupo**: Botones button que tienen clase primary. (especificidad 011)
- **Tercer grupo**: Botones que tienen clase primary. (especificidad 010)

Al contrario que muchos desarrolladores piensan, **en CSS no se fusionan los estilos** al ser (al fin y al cabo) el mismo elemento, **sino que los selectores más específicos son los que tendrán prioridad, independientemente del orden**:

```css
button[class].primary {
  color: red;
}

button.primary {
  background: #34a;
  border: 2px outset #6381db;
  color: #fff;
  padding: 5px 10px;
  border-radius: 6px;
}

.primary {
  margin: 20px;
  color: gold;
}
```

```html
<button class="primary">Click me!</button>
```

En este caso, se aplicará primero el último grupo (al ser el menos especifico), luego se aplicarán los estilos del segundo grupo, sobreescribiendo el color blanco por el color dorado. Por último, el primer bloque sobreescribirá con el color rojo.

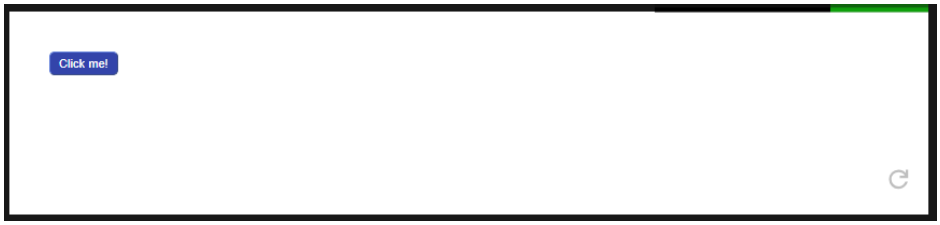
Modifiquemos un poco el ejemplo anterior, y vamos a añadir el primer grupo en una capa llamada texts:

```css
@layer texts {
  button[class].primary {
    color: red;
  }
}

button.primary {
  background: #34a;
  border: 2px outset #6381db;
  color: #fff;
  padding: 5px 10px;
  border-radius: 6px;
}

.primary {
  margin: 20px;
  color: gold;
}
```

```html
<button class="primary">Click me!</button>
```



- Ahora, observa que la capa texts creada será la primera en procesarse por el navegador.
- Lo primero que hará es agrupar todos los estilos en dicha capa, calcular sus especificidades y aplicarlas.
- Luego, buscará si existen otras capas diferentes para procesarlas. Si no existen más, agrupará el resto de los estilos fuera de capas en una capa anónima y los aplicará después de los anteriores.

De esta forma, hemos cambiado el comportamiento que explicamos al principio por algo que, unido a la posibilidad de agrupar en diferentes capas y ordenar mediante la regla @layer se convierte en un recurso muy potente para organizar código por parte de los desarrolladores.

Capas CSS anidadas

Dentro de las capas de CSS, también es posible crear capas dentro de otras. Para ello, solo tenemos que utilizar la regla @layer dentro de otra regla @layer, como se puede ver a continuación:

```
@layer base {
  @layer reset {
    body {
      margin: 0;
      box-sizing: border-box;
    }
  }
}
```

Ten en cuenta que también puedes usar una sintaxis rápida separando los nombres de la capa con puntos. El ejemplo anterior podría definirse también de la siguiente forma:

```
@layer base.reset {
  body {
    margin: 0;
    box-sizing: border-box;
  }
}
```

Importar CSS en una capa

Si ya conocemos la regla @import, sabremos que es posible utilizarla para importar código CSS de ficheros externos e incorporarlo a nuestra página.

Sin embargo, existe una forma de hacerlo añadiéndolo a una capa CSS específica, utilizando la palabra clave layer tras la importación:

```
@import url("framework.css") layer(framework);
```

Esto permitirá que incluso código externo que tengamos separados en diferentes archivos, se pueda colocar directamente en una capa.

Regla @property

La regla @property de CSS es una sencilla pero potente característica de una serie de API denominadas CSS Houdini (en referencia al famoso ilusionista que hacía cosas que parecían imposibles), mediante las cuales puedes realizar ciertas tareas que, en principio, eran imposibles de hacer sólo con CSS directamente en el navegador.

¿Qué es la regla @property?

La regla @property nos permite indicar al navegador el tipo de dato, así como algunos datos relacionados, que tiene una variable CSS.

Si conoces la idea base de Typescript, que es dotar de una comprobación de tipos a Javascript, la idea de la regla @property es muy similar.

¿Para qué podemos necesitar esto en CSS?

Existe un caso particular donde se ve muy claro su utilización.

Imagina que queremos crear una animación para mover un elemento, y el valor que vamos a animar es un tamaño que está guardado en una variable CSS.

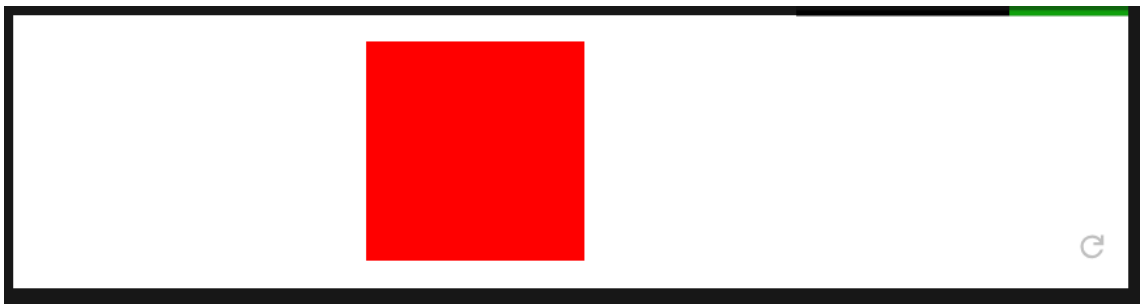
El navegador desconoce el tipo de dato que está guardado en las variables CSS, por lo que no aplicará la animación, sino que saltará de golpe entre el primer valor y el último:

```css
.element {
  --x: 0;

  width: 200px;
  height: 200px;
  background: red;
  animation: move 2s alternate infinite;
  translate: var(--x) 0;
}

@keyframes move {
  to { --x: 300px; }
}
```

```html
<div class="element"></div>
```



Sin embargo, con la regla @property podemos indicarle específicamente de qué tipo de dato se trata, y que actúe como corresponde, entendiendo que esa variable contendrá un tamaño, y como consecuencia, soportando la animación:

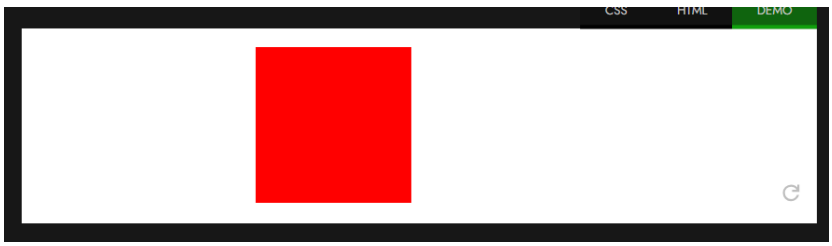
```css
.element {
  --x: 0;

  width: 200px;
  height: 200px;
  background: red;
  animation: move 2s alternate infinite;
  translate: var(--x) 0;
}

@keyframes move {
  to { --x: 300px; }
}

@property --x {
  syntax: "<length>";
  inherits: true;
  initial-value: 0
}
```

```html
<div class="element"></div>
```



Además, ofrece otras ventajas que iremos explicando a continuación.

Sintaxis de la regla @property

Como hemos visto en el ejemplo anterior, para establecer una regla @property debemos establecer la regla seguida de la variable CSS sobre la cuál va a actuar.

En su interior debemos establecer tres propiedades:

Propiedad	Descripción
syntax	Indica la sintaxis (el tipo de dato) de la variable.
inherits	Indica si la variable CSS debe heredarse o no. Valores true o false.
initial-value	Indica cual es el valor por defecto de la variable CSS.

Así pues, veamos otro ejemplo de declaración de @property en acción. En este caso, hemos establecido un tipo de dato de color, para que se pueda animar el color:

```css
@property --color {
  syntax: "<color>";
  inherits: false;
  initial-value: red;
}
```

Como ves, muy sencillo.

Tipos de datos CSS de @property

Los valores posibles de **la propiedad syntax** deben colocarse entre signos angulares **< y >** (como si fuera una etiqueta HTML) y podemos utilizar cualquiera de los siguientes:

Sintaxis posibles	Descripción
`<length>`	Indicamos una distancia o tamaño en una unidad CSS, ya sea absoluta, relativa o de viewport.
`<length-percentage>`	Permite tanto las unidades anteriores como los valores de porcentaje.
`<percentage>`	Indica sólo y exclusivamente valores de porcentajes.
`<angle>`	Permite indicar ángulos, con unidades como `deg` o `turn`, entre otras.
`<time>`	Indica valores de tiempo, como por ejemplo, `s` o `ms`.
`<resolution>`	Permite indicar valores de resolución, muy útiles en media queries, como `dpi`, `dppx` u otros.
`<integer>`	Indica valores numéricos enteros, ya sean positivos o negativos.
`<number>`	Permite tanto los valores anteriores, como valores decimales.
`<color>`	Permite indicar colores CSS, mediante sintaxis hexadecimal, rgb() u otras.
`<custom-ident>`	Valores personalizados por el usuario, similar a un string: nombre de animaciones, propiedades, etc.
`<url>`	Indica una URL mediante la función `url()` de CSS.
`<image>`	Permite indicar los valores anteriores de `url()` o gradientes CSS.
`<transform-function>`	Indica una función de transformación como `translate()`, `scale()` o similares.
`<transform-list>`	Permite una combinación de varias funciones de transformación de las anteriores.

Reglas de @font-feature-values

Las reglas que vemos a continuación, son reglas CSS especiales que sólo sirven dentro de la regla @font-feature-values mencionada anteriormente, para indicar la activación/desactivación de características especiales en tipografías concretas.

Entre ellas, podemos encontrar las siguientes reglas:

Regla	Descripción
`@stylistic`	Estalece características de estilo alternativas individuales.
`@historical-forms`	Establece glifos históricos.
`@styleset`	Establece un pack de características de estilo alternativas.
`@character-variant`	Establece variaciones en carácteres
`@swash`	Establece decoraciones artísticas (adornos tipográficos)
`@ornaments`	Establece glifos con adornos
`@annotation`	Establece formas notacionales de glifos

Ciertas tipografías tienen una serie de características avanzadas que pueden existir o no, dependiendo de la tipografía. Dichas características permiten realizar variaciones en las tipografías como las siguientes:

Característica	Descripción
Ligadura	Unión visual de 2 carácteres. Muy usada en programación, matemáticas e idiomas.
Posicionamiento	Colocación de carácteres de forma similar a los subíndices o superíndices.
Mayúsculas/Minúsculas	Variaciones relacionadas con las mayúsculas y minúsculas.
Variaciones	Se trata de grupos de variaciones visuales con un propósito concreto.

Veamos detalladamente cada una de estas características:

- **Ligaduras**: Es posible activar una característica llamada ligadura que no es más que la unión de ciertos caracteres cambiando su aspecto visual. Esto se puede hacer con la propiedad **font-variant-ligatures** , y dependiendo del valor específico, activaremos uno u otro tipo de ligadura.

  Existen las siguientes:
  - Ligaduras normales: Se indica con el valor common-ligatures . Se suele observar en valores como fi , fl , ff ...
  - Ligaduras discrecionales: Se indica con el valor discretionary-ligatures . Se suele observar en valores como ch , sp , st ...
  - Ligaduras históricas: Se indica con el valor historical-ligatures . Se aplican para mostrar ligaduras heredadas de escritura manual.

- **Posicionamiento de carácteres**: Es posible activar una variación para indicar texto como subíndice o superíndice, similar a como se hace con las etiquetas HTML <sub> y <super> .

- **Mayúsculas y minúsculas**: Es posible realizar ciertas variaciones como versalitas o similares, aplicadas sólo a minúsculas, sólo a mayúsculas o a ambas.

- **Variaciones numéricas**: Con la propiedad font-variant-numeric podemos establecer ciertas variaciones como representar números con estilo antiguo, diferentes proporciones, como fracciones "gráficas" o incluso indicar que el cero tenga una raya o no.

- **Variaciones alternativas**: Con la propiedad font-variant-alternates se pueden indicar una o múltiples variaciones visuales, donde se pueden mencionar las siguientes:

  - **historical-forms:** Muestra glifos históricos (heredados de épocas pasadas)

- **stylistic(id):** Muesta características de estilo alternativas de forma individual.
- **styleset(id):** Muestra un conjunto de características de estilo alternativas (creados para trabajar juntos).
- **character-variant(id):** Muestra ciertas variaciones en carácteres concretos.
- **swash(id):** Muestra decoraciones artísticas en los carácteres como adornos tipográficos.
- **ornaments(id):** Muestra glifos con adornos en lugar del caracter predeterminado.
- **anotation(id):** Muestra formas notacionales de glifos (glifos abiertos, cajas redondeadas, etc...)

Variaciones en tipografías

Estas características pueden ser activadas o desactivadas desde CSS, por medio de una de las siguientes propiedades (o de forma alternativa, utilizando una propiedad a bajo nivel que veremos más adelante):

Propiedad/Valores	Significado
font-variant-ligatures	Establece un estilo de ligadura
	normal | none | common-ligatures | no-common-ligatures | discretionary-ligatures no-discretionary-ligatures | historical-ligatures | no-historical-ligatures contextual | no-contextual
font-variant-position	Establece estilo subscript (subíndice) o superscript (superíndice)
	normal | sub | super
font-variant-caps	Establece una variación de versalitas
	normal | small-caps | all-small-caps | petite-caps all-petite-caps | unicase | titling-caps
font-variant-numeric	Establece variaciones numéricas
	normal | lining-nums | oldstyle-nums | proportional-nums | tabular-nums diagonal-fractions | stacked-fractions | ordinal | slashed-zero
font-variant-alternates	Establece variaciones visuales
	normal | historical-forms | stylistic(id) | styleset(id) | character-variant(id) swash(id) | ornaments(id) | annotations(id)
font-variant-east-asian	Permite controlar los glifos de textos orientales
	normal | jis78 | jis83 | jis90 | jis04 simplified | traditional | full-width | proportional-width | ruby
font-kerning	Indica quién debe ajustar el kerning
	auto | normal | none

Ten en cuenta que, en la mayoría de ellas, se puede utilizar la regla **@font-feature-values** para personalizar como utilizar la característica en cuestión (la cuál debe existir y estar definida en la tipografía). Veamos un ejemplo:

```
@font-feature-values 'bookmania' {
    @swash {
        flourish: 1;
    }
}

.texto-alternativo {
    font-variant-alternates: swash(flourish);
}
```

Características a bajo nivel

La propiedad **font-feature-settings es una propiedad a bajo nivel para indicar como debe comportarse una tipografía respecto a sus características especiales**. Sólo debe usarse cuando no nos sirva otra propiedad con la que modificar dicha característica, la cuál normalmente se encuentra en la lista del apartado anterior.

La sintaxis de esta propiedad es la que podemos ver a continuación (se pueden combinar varias características separando por comas):

```
p {
    /* La característica "feat" está activada. */
    font-feature-settings: "feat" 1;
    /* Otra opción equivalente a la anterior. */
    font-feature-settings: "feat" on;
    /* La característica "feat" está desactivada. */
    font-feature-settings: "feat" 0;
    /* Otra opción equivalente a la anterior. */
    font-feature-settings: "feat" off;
}
```

Sin embargo, la característica «feat» no existe, es sólo un ejemplo teórico. Cada tipografía tiene un cierto número de características especiales (hay tipografías que tienen varias, y tipografías que no tienen ninguna). Veamos una lista de las características que existen:

Valor OTF	Significado	Propiedad equivalente	Valor concreto
Características de ligaduras			
"liga" 1	Ligaduras estándar (también clig)	font-variant-ligatures	common-ligatures
"dlig" 1	Ligaduras discrecionales	font-variant-ligatures	discretionary-ligatures
"hlig" 1	Ligaduras históricas	font-variant-ligatures	historical-ligatures
"calt" 1	Ligaduras contextuales alternativas	font-variant-ligatures	contextual
Características numéricas			
"ordn" 1	Marcas ordinales	font-variant-numeric	ordinal
"zero" 1	Raya del cero	font-variant-numeric	slashed-zero
"onum" 1	Figuras de estilo antiguo	font-variant-numeric	oldstyle-nums
"lnum" 1	Números alineados	font-variant-numeric	lining-nums
"pnum" 1	Números proporcionales	font-variant-numeric	proportional-nums
"tnum" 1	Figuras tabulares (mismo tamaño)	font-variant-numeric	tabular-nums
"frac" 1	Fracciones diagonales	font-variant-numeric	diagonal-fractions
"afrc" 1	Fracciones apiladas	font-variant-numeric	stacked-fractions
Características de posición			
"sups" 1	Superíndices	font-variant-position	super
"subs" 1	Subíndices	font-variant-position	sub
Características de mayúsculas/minúsculas			
"smcp" 1	Versalitas en minúsculas	font-variant-caps	small-caps
"c2sc" 1	Versalitas en mayúsculas y minúsculas	font-variant-caps	all-small-caps
"pcap" 1	Capitaliza minúsculas	font-variant-caps	petite-caps
"c2pc" 1	Capitaliza mayúsculas y minúsculas	font-variant-caps	all-petite-caps
"unic" 1	Versalitas en mayúsculas	font-variant-caps	unicase
"titl" 1	Mayúsculas para titulares	font-variant-caps	titling-caps
Características alternativas			
"swsh" 1	Decoraciones swash	font-variant-alternates	swash()
"hist" 1	Glifos históricos	font-variant-alternates	historical-forms
"salt" 1	Carácteres con estilos alternativos	font-variant-alternates	stylistic()
"ss01" 1	Conjunto de estilos alternativos	font-variant-alternates	styleset(01)
"ss02" 1	Segundo conjunto de estilos alternativos	font-variant-alternates	styleset(02)
"ss03" 1	Tercer conjunto de estilos alternativos	font-variant-alternates	styleset(03)
"cv01" 1	Estilo alternativo para carácteres	font-variant-alternates	character-variant(01)
"ornm" 1	Adornos como fleurons o dingbats	font-variant-alternates	ornaments()
"nalt" 1	Dígitos circulares, carácteres invertidos...	font-variant-alternates	annotation()
"kern" 1	Kerning de tipografías	font-kerning	normal

Atajo de características

Tradicionalmente, la propiedad **font-variant** sólo permitía los valores normal o **small-caps** . Sin embargo, las nuevas propuestas de CSS permiten utilizar esta propiedad como propiedad de atajo para todas las características que hemos visto en el apartado anterior.

Por lo tanto, pasamos de este esquema antiguo (soportado en la mayoría de los navegadores):

Propiedad	Valor	Significado
font-variant	**normal** | small-caps	Indica si utilizar versalitas o no

Al esquema moderno, que es el siguiente:

```
p {
  /* font-variant: <ligatures> <caps> <numeric> <east-asian> <position> */
}
```

En él, podemos utilizar los valores de cada propiedad individual: **font-variant-ligatures , font-variant-caps** , etc... en forma de atajo.

Regla @supports

Existe una regla CSS denominada **@supports** que permite establecer fragmentos de código CSS condicionales, aplicando estilos CSS sólo cuando se cumplen ciertas condiciones y restricciones.

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.

Formato	Descripción
@supports (condición)	Aplica los estilos si se cumple la condición.
@supports not (condición)	Aplica los estilos si no se cumple la condición.
@supports (condición1) and (condición2)	Aplica los estilos si se cumplen las diferentes condiciones.
@supports (condición1) or (condición2)	Aplica los estilos si se cumple una de las dos condiciones.

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

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

Observa que en este ejemplo, **se definen unos 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, es decir, que tenga soporte de Grid CSS. Podemos hacer esto mismo con cualquier otra propiedad CSS, y utilizar la regla @supports para crear códigos condicionales.

Reglas compuestas

Sin embargo, 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 flexbox, siempre en el caso de que el navegador no soporte Grid CSS pero si soporte 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.

Reglas con fallbacks

En lugar de establecer reglas negadas compuestas, es mejor utilizar un enfoque donde establezcas unos estilos generales, que se sobrescriben 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 CSS, 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 CSS pero si **Flexbox**.

Regla @when / @else

@when / @else	Establece unos estilos si se cumple la condición y otros si no se cumple.	⚠ Aún sin soporte.

Las reglas condicionales @when / @else nos permiten aplicar estilos en función de una condición, de forma que, si la condición se cumple, se aplican los estilos de la regla @when, y si no se cumple, se aplican los estilos de la regla @else.

Aunque ya teníamos la opción de utilizar las media queries para utilizar condiciones a la hora de aplicar reglas, el uso de estas nuevas reglas nos permiten agrupar queries mutuamente excluyentes de forma que se apliquen los estilos de la primera regla que se cumpla, y no se apliquen los estilos de las demás reglas.

```
@when (min-width: 600px) {
  .container {
    display: grid;
  }
}
@else supports(caret-color: pink) {
  .container {
    display: block;
  }
}
@else {
  .container {
    display: flex;
  }
}
```

Regla @function

`@function`	Crea una función CSS que dada una entrada, produce una salida de CSS.	⚠ Aún sin soporte.

Regla @mixins / @apply

`@mixins` / `@apply`	Crea un grupo de propiedades CSS que se pueden aplicar o reutilizar.	⚠ Aún sin soporte.

Contadores CSS

Si conocemos los estilos que es posible darle a una lista con CSS es posible que nos interese conocer la posibilidad de crear contadores personalizados con CSS y utilizarlos en nuestras páginas para contar diferentes elementos de la misma, además de crear sistemas de numeración personalizados y adaptados a nuestros criterios.

Propiedades de contador

Para empezar, en CSS podemos crear un contador CSS.

Básicamente, lo único que tenemos que hacer es asignarle un id o nombre, que nos permitirá hacer referencia a él posteriormente.

Veamos que propiedades podemos utilizar para esto:

Propiedades	Valores	Descripción
`counter-reset`	`id valor_inicial` | **none**	Resetea el contador `id` al `valor_inicial` (`0` por defecto).
`counter-increment`		Incrementa un contador, indicándole su `id` de referencia.
`counter-set`	`id valor_inicial` | **none**	Similar a `counter-reset`, pero no realiza incremento si existe.

En primer lugar, tenemos las propiedades **counter-reset y counter-increment** **que no hacen otra cosa más que resetear a cero o incrementar un contador CSS**, indicándole el nombre o id del mismo.

La función counter()

Sin embargo, necesitamos mostrarlo de alguna forma o sería inútil.

Para ello, podemos utilizar la función CSS counter(), que se suele utilizar junto a la propiedad content en pseudoelementos ::before o ::after.

A dicha función, se le debe pasar de forma obligatoria el id del contador, y de forma opcional el tipo de contador que queremos utilizar, que, de no hacerlo, tomará por defecto el tipo decimal:

Función	Descripción
counter(id , type)	Usa el contador con id aplicando un estilo type.

Así pues, veamos un ejemplo en acción con lo que hemos visto hasta ahora.

El siguiente ejemplo incorpora 3 elementos con clase item que queremos contabilizar, a pesar de no ser una lista y, además, teniendo contenido de por medio.

```
.item {
  counter-increment: detail-counter;
}

.item::before {
  content: counter(detail-counter) " ";
  font-weight: bold;
  color: red;
}
```

```
<div class="content">
  <p>Esto es un ejemplo de varios detalles:</p>

  <div class="item">Detalle A</div>
  <div>Explicación opcional del detalle.</div>
  <div class="item">Detalle B</div>
  <div class="item">Detalle C</div>

  <p>Otro contenido no relevante.</p>
</div>
```

Esto es un ejemplo de varios detalles:

1 Detalle A
Explicación opcional del detalle.
2 Detalle B
3 Detalle C

Otro contenido no relevante.

Para ello, establecemos un incremento en cada elemento con clase .item y además, inmediatamente antes de ese elemento, colocamos visualmente el contador, en rojo y negrita, seguido de un espacio en blanco.

Observa que el contador lo hemos llamado detail-counter, aunque se podría llamar de cualquier otra forma.

Aunque no se muestre explícitamente, es como si tuvieramos un counter-reset: detail-counter 0 en el elemento .content, que es el que inicializa todo a 0. El contador se inicializa en 0 pero realiza el primer incremento antes de mostrar el valor, por lo que los números contarán a partir del 1.

En la función counter() no indicamos tipo, por lo tanto usará el tipo decimal por defecto.

Si quieres saber que otros tipos existen, echa un vistazo al artículo de Listas CSS.

Creación de contadores

Si necesitamos algún tipo de contador personalizado y no nos bastan con los que incorpora CSS, es posible crearlo y customizarlo a nuestro gusto, mediante la regla **@counter-style**.

Básicamente, te permite crear un tipo de contador, y decirle los símbolos que vamos a utilizar.

Regla	Descripción
`@counter-style nombre`	Crea un tipo de numeración para utilizar en contadores CSS.

Dentro de la regla @counter-style, en la que debemos especificar un nombre para hacer referencia, podemos utilizar varias propiedades.

Veamos un ejemplo sencillo, en el que creamos un sistema de numeración llamado spanish que cuenta los números del uno al tres, colocándole a cada uno el sufijo »:

```
@counter-style spanish {
  system: cyclic;
  symbols: "uno" "dos" "tres";
  suffix: "» ";
}

ul {
  list-style-type: spanish;
}

li::marker {
  color: red;
}
```

```
<ul>
  <li>Primer elemento.</li>
  <li>Segundo elemento.</li>
  <li>Tercer elemento.</li>
</ul>
```

uno» Primer elemento.
dos» Segundo elemento.
tres» Tercer elemento.

Observa que, al establecer un sistema cíclico, una vez terminamos el tres, volvemos a empezar en uno.

Además de system, symbols y suffix existen otras propiedades que se pueden utilizar en las reglas @counter-style, echemos un vistazo y expliquemos para que funcionan y que valores pueden tomar:

Propiedad	Descripción
system	Establece el sistema de repetición al terminar los símbolos. El valor por defecto es symbolic.
symbols	Indica los símbolos de nuestro sistema de numeración, separados por espacios.
additive-symbols	Indica los símbolos de nuestro sistema de numeración de forma acumulativa.
suffix	Indica un sufijo que aparecerá siempre al final.
prefix	Indica un prefijo que aparecerá siempre al principio.
negative	Indica el prefijo y sufijo que tendrían los valores negativos.
range	Limita el ámbito del contador.
pad	Indica la longitud y relleno que usará. Por defecto, el valor es 0 ""
fallback	Indica un tipo de contador de fallback, en caso de que no pueda crearse.
speak-as	Indica como representará el contador un sintetizador de voz.

La propiedad system

Un sistema de numeración puede comportarse de varias formas. Tenemos una propiedad denominada system que permite indicar uno de los siguientes valores y determinar su comportamiento:

Valor	Descripción	Ejemplo
cyclic	Establece un comportamiento cíclico.	A B C -> A, B, C, A, B, C, A...
fixed	Establece una numeración de un número finito de elementos.	A B C -> A, B, C, 4, 5, 6...
symbolic	Estable un patrón repetible y acumulable.	A B C -> A, B, C, AA, BB, CC, AAA...
alphabetic	Estable un patrón alfabético repetible y acumulable.	A B C -> A, B, C, AA, AB, AC, BA, BB, BC, CA...
numeric	Establece un patrón numérico repetible con cero.	A B C -> B, C, BA, BB, BC, CA, CB, CC, BAA, BAB...
additive	Establece un patrón añadiendo símbolos asociados y acumulables.	⚀, ⚁, ⚂, ⚃, ⚄, ⚅, ⚅⚀, ⚅⚁, ⚅⚂, ⚅⚃, ⚅⚄, ⚅⚅, ⚅⚅⚀...
extends	Simplemente extiende un sistema con un sufijo o prefijo.	") " -> 1), 2), 3), 4), 5), 6), 7), 8), 9)...

A continuación, observa la combinación de un system: additive junto a la propiedad additive-symbols para crear combinaciones de símbolos, ya que es ligeramente diferente a las anteriores:

```
@counter-style dice {
  system: additive;
  additive-symbols: 6 ⚅, 5 ⚄, 4 ⚃, 3 ⚂, 2 ⚁, 1 ⚀;
  suffix: " ";
}

ul {
  list-style-type: dice;
  font-size: 2rem;
}

li::marker {
  color: red;
}
```

```html
<ul>
  <li>Primer elemento.</li>
  <li>Segundo elemento.</li>
  <li>Tercer elemento.</li>
  <li>Cuarto elemento.</li>
  <li>Quinto elemento.</li>
  <li>Sexto elemento.</li>
</ul>
```

La propiedad symbols

Mediante la propiedad symbols, o la propiedad additive-symbols, podemos establecer una lista de símbolos para crear nuestro sistema de numeración.

Además de establecer una lista de símbolos, también es posible añadir una lista de imágenes, mediante la función url(imagen.png), por ejemplo.

La propiedad negative

La propiedad negative espera dos parámetros.

El primero de esos parámetros será el string que se añadirá justo ante de un símbolo negativo.

Por otro lado, el segundo parámetro será el string que se añadirá justo después de un símbolo negativo. Veamos dos ejemplos:

```css
/* (2), (1), 0, 1, 2, 3, 4, 5, 6, 7, 8... */
@counter-style negative-paren {
  system: numeric;
  symbols: "0" "1" "2" "3" "4" "5" "6" "7" "8" "9";
  negative: "(-" ")";
}

/* -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8... */
@counter-style negative-minus {
  system: numeric;
  symbols: "0" "1" "2" "3" "4" "5" "6" "7" "8" "9";
  negative: "-" "";
}

ol {
  list-style-type: negative-paren;
}
```

```html
<ol start="2" reversed>
  <li>Elemento dos.</li>
  <li>Elemento uno.</li>
  <li>Elemento cero.</li>
  <li>Elemento menos uno.</li>
  <li>Elemento menos dos.</li>
  <li>Elemento menos tres.</li>
</ol>
```

La propiedad range

Con la propiedad range podemos utilizar el valor auto o uno personalizado.

De indicar el valor auto, se establecerá un rango dependiendo del valor indicado en la propiedad system. Por lo tanto:

- Para sistemas cyclic, numeric o fixed, el rango va desde -∞ a ∞.
- Para sistemas alphabetic o symbolic, el rango va desde 1 a ∞.
- Para sistemas additive, el rango va desde 0 a ∞.
- Para sistemas extends, se adapta al estilo extendido.

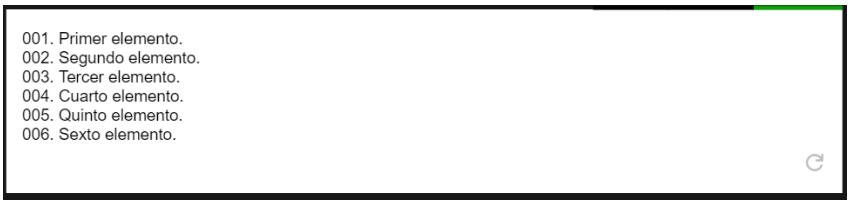
La propiedad pad

Mediante la propiedad pad se puede rellenar con ceros (o con otros carácteres) el sistema de numeración que queramos utilizar. Por ejemplo, para extender el sistema decimal y obtener los siguientes valores: 001, 002, 003, 004, 005, 006, etc... necesitaremos hacer lo siguiente:

```css
@counter-style decimal-zeropad {
  system: extends decimal;
  pad: 3 "0";
}

ul {
  list-style-type: decimal-zeropad;
}
```

```html
<ul>
  <li>Primer elemento.</li>
  <li>Segundo elemento.</li>
  <li>Tercer elemento.</li>
  <li>Cuarto elemento.</li>
  <li>Quinto elemento.</li>
  <li>Sexto elemento.</li>
</ul>
```



Ten en cuenta que **la propiedad pad** permite definir **un primer parámetro que será el número de dígitos que debe tener el número del contador, mientras que el segundo parámetro será el relleno que utilizará a la izquierda hasta llegar a ese número de dígitos indicado anteriormente**.

La propiedad fallback

La propiedad fallback simplemente determina el tipo de sistema que va a utilizarse en el caso de que no se pueda usar algún otro que haya sido indicado.

Por defecto, si no se especifica, utilizará decimal.

La propiedad speak-as

La propiedad speak-as intenta garantizar que los contadores se puedan no solo ver visualmente a través de un navegador, sino reproducir a través de sintetizadores de voz y sistemas equivalentes.

Con esta propiedad puedes determinar cómo se comportará al leerlo, indicando uno de los siguientes valores:

Valor	Descripción
auto	Si system es alphabetic, usa spell-out. Si es cyclic, usa bullets. En otro caso, usa numbers.
bullets	Lee la lista como si fuera una lista sin orden.
numbers	Lee la lista como si fuera una lista numerada.
words	Lee la lista como un valor textual. Si contiene imágenes, las lee como numbers.
spell-out	Lee la lista como un valor textual, letra a letra. Si no reconoce símbolos, los lee como numbers.
nombre	Lee la lista como se declara, sino actua como auto.

Esto proporcionará una mejor accesibilidad a nuestras listas o contadores de cara a sintentizadores de voz o lectores de accesibilidad.

Regla @color-profile

La regla at de CSS define y nombra un perfil de color que luego se puede usar en la función para especificar un color.**@color-profile color()**

Sintaxis

```CSS
@color-profile --swop5c {
  src: url("https://example.org/SWOP2006_Coated5v2.icc");
}
.header {
  background-color: color(--swop5c 0% 70% 20% 0%);
}
```

Sintaxis formal

```
@color-profile =
  @color-profile [ <dashed-ident> | device-cmyk ] { <declaration-list> }
```

Descriptores

src

Especifica la URL de la que recuperar la información del perfil de color.

rendering-intent

Si el perfil de color contiene más de una intención de representación, este descriptor permite eleccionar uno como el que se utilizará para definir cómo asignar el color a gamas más pequeñas que las que se definen en este perfil.

Si se utiliza, debe ser una de las siguientes palabras clave:

relative-colorimetric

La colorimetría relativa al medio es necesaria para dejar sin cambios los colores de origen que se encuentran dentro de la gama de medios de destino en relación con los puntos blancos del medio respectivo. Los colores de origen que están fuera de la gama media de destino se asignan a colores en el límite de la gama mediante una variedad de métodos diferentes.

absolute-colorimetric

Se requiere colorimetría absoluta ICC para dejar los colores de origen que se encuentran dentro de la gama media de destino sin cambios en relación con el blanco adoptado (un difusor reflectante perfecto). Los colores de origen que están fuera de la gama media de destino se asignan a colores en el límite de la gama mediante una variedad de métodos diferentes.

perceptual

Este método suele ser la opción preferida para las imágenes, especialmente cuando existen diferencias sustanciales entre el origen y el destino (como una imagen en pantalla

reproducida en una impresión reflejada). Toma los colores de la imagen de origen y vuelve a optimizar la apariencia del medio de destino utilizando métodos propietarios.

saturation

Esta opción se creó para preservar la saturación relativa (croma) del original y mantener puros los colores sólidos. Sin embargo, experimentó problemas de interoperabilidad como la intención de percepción.

Regla @namespace

@namespace es una **regla que define XML namespace** a ser usados en una hoja de estilos CSS.

Regla @starting-style

En fase experimental.

Regla @starting-style

Obsoleta.

Documento para estudiantes extraído de distintas fuentes de internet

BIBLIOWEB:

https://developer.mozilla.org/es/docs/Web/CSS/Reference#keyword_index

https://lenguajecss.com/css/reglas-css/que-son-reglas-css/