

## Tema 2. Módulos de diseño en CSS, Media Queries y Problemas

### Índice

<b>1. Módulos de diseño en CSS</b>	<b>2</b>
1.1. Modelo de caja normal (Block e Inline)	2
1.2. Floats (Flotantes)	3
1.3. Positioning (Posicionamiento)	4
1.4. Flexbox (Flexible Box Layout)	5
1.5. CSS Grid Layout	6
1.6. Multi-column Layout	7
1.7. CSS Table Layout	8
1.8. Exclusions y Shapes	10
<b>2. Media Queries en CSS</b>	<b>12</b>
<b>3. Variables CSS (Custom Properties)</b>	<b>14</b>
<b>4. Accesibilidad y UX</b>	<b>15</b>
4.1. Contraste y legibilidad	15
4.2. Preferencias del usuario	16
4.3. Uso de etiquetas semánticas	16
4.4. Navegación por teclado	17
4.5. Compatibilidad con lectores de pantalla	17
4.6. Diseño responsivo e inclusivo	17
4.7. Experiencia de usuario (UX)	18
4.8. Conclusión	18
<b>5. Responsive avanzado</b>	<b>19</b>
5.1. Estrategia Mobile First	19
5.2. Container Queries	19
5.3. Unidades relativas y modernas	20
5.4. Grid Layout responsivo	20
5.5. Patrones avanzados de responsive	20
5.6. Conclusión	21
<b>6. Calidad y mantenibilidad</b>	<b>22</b>
6.1. Metodologías de organización	22
6.2. Buenas prácticas en CSS	23
6.3. Conclusión	24
<b>7. Tendencias actuales</b>	<b>25</b>
7.1. Subgrid	25
7.2. Nesting en CSS	25
7.3. Logical Properties	26
7.4. Otras tendencias emergentes	27
7.5. Impacto en la práctica profesional	27
7.6. Conclusión	28

<b>8. Ejercicios con solución</b>	<b>29</b>
8.1. Ejercicio Especificidad y colisión de estilos . . . . .	29
8.2. Ejercicio Selectores combinados y herencia . . . . .	29
8.3. Ejercicio Modelo de caja y <b>box-sizing</b> . . . . .	29
8.4. Ejercicio Unidades y accesibilidad tipográfica . . . . .	30
8.5. Ejercicio Paleta y contraste . . . . .	30
8.6. Ejercicio <b>display</b> vs <b>visibility</b> y flujo . . . . .	32
8.7. Ejercicio Responsive con media queries . . . . .	33
8.8. Ejercicio Transición y estado <b>:hover/:</b> focus-visible . . . . .	34
8.9. Ejercicio Flexbox: alineación de tarjetas . . . . .	34
8.10. Ejercicio Grid: layout holy-grail simple . . . . .	36
<b>9. Ejercicios propuestos</b>	<b>38</b>
9.1. Selector <b>:nth-child</b> complejo . . . . .	38
9.2. Animaciones con <b>@keyframes</b> . . . . .	39
9.3. Pseudo-elementos avanzados . . . . .	39
9.4. Variables CSS y herencia . . . . .	40
9.5. Grid con áreas solapadas (1.5) . . . . .	41
9.6. Flexbox con <b>order</b> y <b>grow</b> . . . . .	42
9.7. Clipping y máscaras . . . . .	42
9.8. Tipografía responsiva avanzada . . . . .	43
9.9. Animación con variables CSS . . . . .	44
9.10. Accesibilidad y media queries avanzadas . . . . .	45

# 1. Módulos de diseño en CSS

Los módulos de diseño en CSS son distintos sistemas que definen cómo se organizan, alinean y distribuyen los elementos dentro de una página web. Cada módulo ofrece un enfoque específico: el modelo de caja normal (block e inline) (1.1) establece el flujo básico de los elementos; los floats (1.2) y el positioning (1.3) permiten sacarlos de ese flujo para colocarlos con mayor precisión; Flexbox (1.4) organiza de forma flexible en una dimensión (filas o columnas); Grid (1.5) crea estructuras bidimensionales de filas y columnas; Multi-column Layout (1.6) divide texto en columnas al estilo de periódicos; y Table Layout (1.7) reutiliza el comportamiento de tablas. Incluso existen módulos experimentales como Exclusions y Shapes (1.8) que permiten flujos alrededor de formas no rectangulares. En conjunto, estos módulos constituyen la base para construir interfaces modernas, adaptables y accesibles.

## 1.1. Modelo de caja normal (Block e Inline)

**Concepto** Es el modelo de diseño por defecto en CSS. Cada elemento HTML genera una **caja** que se integra en el flujo normal de la página. Según su tipo, esa caja se comporta como **block** o **inline**.

### Elementos de tipo block

**Comportamiento** Ocupan todo el ancho disponible de su contenedor y fuerzan un salto de línea antes y después.

**Ejemplos** <div>, <p>, <section>, <h1>, etc.

**Propiedades aplicables** Se pueden ajustar dimensiones (**width**, **height**), márgenes y padding en los cuatro lados.

### Elementos de tipo inline

**Comportamiento** Se colocan en la misma línea que el texto y ocupan únicamente el espacio de su contenido.

**Ejemplos** <span>, <a>, <em>, <strong>, etc.

**Propiedades aplicables** Aceptan márgenes y padding horizontales, pero los verticales no desplazan el flujo de línea.

**Modelo de caja (Box Model)** Cada caja (block o inline) se compone de:

**Content** El área donde se muestra el texto o imagen.

**Padding** Espacio interior entre el contenido y el borde.

**Border** Línea que rodea la caja.

**Margin** Espacio exterior entre la caja y otras cajas.

**Observación** Aunque cada elemento tiene un tipo de caja por defecto, se puede cambiar con la propiedad **display**, por ejemplo: `span { display: block; }`, `div { display: inline; }`.

```
1  /* Elemento block */
2  p {
3      display: block;
4      width: 300px;
5      margin: 10px auto;
6      padding: 15px;
7      border: 2px solid black;
8      background: #f0f0f0;
9  }
```

```

11  /* Elemento inline */
12  a {
13      display: inline;
14      color: blue;
15      padding: 5px; /* solo afecta horizontalmente */
16      text-decoration: underline;
17  }

```

## 1.2. Floats (Flotantes)

**Concepto** La propiedad `float` en CSS permite sacar un elemento del flujo normal del documento y alinearlo a un lado del contenedor (izquierda o derecha). El contenido posterior fluye alrededor del elemento flotado, como ocurría tradicionalmente con imágenes en periódicos.

**Uso original** Diseñada para colocar imágenes con texto a su alrededor:

```

1  img {
2      float: right;
3      margin: 0 0 1em 1em;
4  }

```

De este modo, el texto se acomoda automáticamente en el espacio libre a la izquierda.

**Uso histórico en layouts** Antes de Flexbox (1.4) y Grid (1.5), los desarrolladores web usaban `float` para crear columnas y estructuras de página. Se combinaba con anchos fijos (`width`) y la técnica de `clearfix` para forzar a que los contenedores incluyeran elementos flotados. Ejemplo típico: menús laterales a la izquierda y contenido principal a la derecha.

### Problemas comunes

**Clearfix** Los elementos flotados no expanden la altura del contenedor padre, lo que obliga a usar `clear: both;` o hacks como `::after{content:;;display:block;clear:both;}`.

**Rígido** Difícil de adaptar a pantallas responsivas, pues requería mucho cálculo manual de anchos.

**No semántico** Se usaba con fines de maquetación, aunque no fue diseñado para ello.

**Estado actual** Hoy en día, el uso de floats para layout está en desuso porque Flexbox (1.4) y Grid (1.5) ofrecen soluciones más potentes y semánticas. Sin embargo, `float` sigue siendo útil en casos puntuales, como rodear imágenes con texto en artículos.

```

1  .caja {
2      float: left;
3      width: 40%;
4      margin: 10px;
5      background: #f0f0f0;
6      padding: 10px;
7  }
8  .clearfix::after {
9      content: "";
10     display: table;
11     clear: both;
12 }

```

Aquí se crean dos cajas flotadas a izquierda, y con la clase `clearfix` el contenedor se adapta a su altura.

### 1.3. Positioning (Posicionamiento)

**Concepto** El posicionamiento en CSS permite controlar con precisión dónde aparece un elemento en la página. A diferencia de otros módulos de layout (como Flexbox (1.4) o Grid (1.5)), no distribuye automáticamente el espacio, sino que mueve un elemento en relación con un punto de referencia.

#### Valores principales de position

**static** Valor por defecto. El elemento sigue el flujo normal del documento sin posicionamiento especial.

**relative** Permite desplazar un elemento respecto a su posición original, usando **top**, **left**, **right**, **bottom**. El espacio original se mantiene reservado en el flujo.

**absolute** El elemento se posiciona en relación con el contenedor más cercano que no sea **static**. Si no lo hay, se referencia al `<html>` o `<body>`. Se sale del flujo normal y no reserva espacio.

**fixed** Similar a **absolute**, pero la referencia es siempre la ventana del navegador (viewport). Se mantiene fijo al hacer scroll. Ideal para cabeceras, menús flotantes o botones de acción.

**sticky** Híbrido entre **relative** y **fixed**. El elemento se comporta como relativo hasta que se alcanza un umbral (ej. **top:0**), y a partir de ahí queda fijo mientras esté dentro de su contenedor padre.

#### Propiedades asociadas

**top**, **left**, **right**, **bottom** Desplazan el elemento según el modo de posicionamiento.

**z-index** Controla la superposición de elementos en el eje Z (qué queda encima o debajo).

**Casos típicos** **Overlays** Cuadros de diálogo o modales centrados con **absolute** o **fixed**.

**Cabeceras fijas** Menús superiores con **fixed** o **sticky**.

**Badges o etiquetas** Elementos pequeños en esquinas de tarjetas con **absolute**.

**Limitaciones** No es un sistema de layout completo porque no distribuye automáticamente los elementos ni gestiona la adaptabilidad. Se usa en conjunto con otros módulos (Flexbox, Grid) para crear interfaces completas.

```
1  .relativo {
2      position: relative;
3      top: 10px;
4      left: 20px;
5  }
6
7  .absoluto {
8      position: absolute;
9      top: 0;
10     right: 0;
11 }
12
13 .fijo {
14     position: fixed;
15     bottom: 10px;
16     right: 10px;
17 }
18
19 .pegajoso {
20     position: sticky;
21     top: 0;
```

```
22 | background: yellow;
23 | }
```

## 1.4. Flexbox (Flexible Box Layout)

**Concepto** Flexbox es un módulo de diseño unidimensional (una fila o una columna). Está pensado para distribuir espacio entre elementos dentro de un contenedor y para alinear dichos elementos de forma precisa, incluso cuando sus tamaños son dinámicos o desconocidos.

**Contenedor Flex** Se crea con `display: flex` o `display: inline-flex`. Todos los hijos directos de este contenedor se convierten en ítems flexibles. Permite controlar:

**Dirección** Con `flex-direction: row | column | row-reverse | column-reverse`.

**Distribución** Con `justify-content` (a lo largo del eje principal).

**Alineación** Con `align-items` (a lo largo del eje transversal).

**Flexibilidad** Con `flex-grow`, `flex-shrink`, `flex-basis` en los hijos.

### Ejes en Flexbox

**Eje principal** Determinado por `flex-direction`. Puede ser horizontal (fila) o vertical (columna).

**Eje transversal** Perpendicular al eje principal. Sirve para la alineación secundaria.

**Propiedades principales en el contenedor `flex-direction`** Define la orientación de los ítems (fila o columna).

**`flex-wrap`** Permite que los ítems se ajusten a varias líneas (`wrap`) o se mantengan en una sola línea.

**`justify-content`** Controla la distribución en el eje principal (`flex-start`, `center`, `space-between`, `space-around`, `space-evenly`).

**`align-items`** Alinea los ítems en el eje transversal (`flex-start`, `center`, `flex-end`, `stretch`).

**`align-content`** Alinea varias líneas de ítems cuando hay `flex-wrap`.

### Propiedades principales en los ítems

**`flex-grow`** Define cuánto puede crecer un ítem en relación a los demás.

**`flex-shrink`** Define cuánto puede encogerse un ítem si falta espacio.

**`flex-basis`** Tamaño inicial antes de distribuir el espacio extra.

**`order`** Permite cambiar el orden visual de los ítems sin alterar el HTML.

**`align-self`** Sobrescribe `align-items` para un ítem específico.

### Casos de uso

**Menús horizontales o verticales** Con distribución equitativa de ítems.

**Barras de navegación** Ítems alineados al centro o a los extremos.

**Tarjetas adaptables** Que crecen o se encogen según el espacio disponible.

**Componentes centrados** Elementos perfectamente centrados en un contenedor.

### Ventajas

**Flexibilidad** Se adapta a pantallas de diferentes tamaños sin cálculos manuales.

**Simplicidad** Resuelve problemas de alineación que antes requerían hacks.

**Compatibilidad** Está soportado en todos los navegadores modernos.

**Limitaciones** Al ser un sistema unidimensional, no gestiona filas y columnas simultáneamente. Para layouts complejos se recomienda usar CSS Grid en combinación con Flexbox.

```
1  .contenedor {
2      display: flex;
3      flex-direction: row;
4      justify-content: space-between;
5      align-items: center;
6  }
7
8  .item {
9      flex-grow: 1;          /* todos ocupan mismo espacio */
10     margin: 5px;
11     background: lightgray;
12     text-align: center;
13 }
14
15 .item.destacado {
16     flex-grow: 2;          /* este ocupa el doble de espacio */
17     order: -1;            /* aparece antes que los demás */
18 }
```

## 1.5. CSS Grid Layout

**Concepto** CSS Grid es un sistema de diseño bidimensional, capaz de organizar elementos en filas y columnas simultáneamente. A diferencia de Flexbox (1.4), que trabaja en una sola dimensión, Grid permite construir maquetaciones completas con gran precisión.

**Contenedor Grid** Se crea con `display: grid` o `display: inline-grid`. Todos los hijos directos se convierten en ítems de la rejilla, que pueden colocarse en celdas definidas o en áreas personalizadas.

### Definición de filas y columnas

**grid-template-columns, grid-template-rows** Definen el número y tamaño de las pistas.

**fr** Unidad específica de Grid que representa una fracción del espacio disponible.

**repeat()** Permite declarar columnas o filas repetidas de forma compacta.

**minmax(min, max)** Define un rango de tamaños entre un mínimo y un máximo.

### Áreas de Grid

**grid-template-areas** Asigna nombres a regiones de la cuadrícula, lo que facilita posicionar ítems mediante `grid-area`.

**Ventaja** Ofrece una forma semántica y legible de construir layouts.

### Colocación de ítems

**grid-column, grid-row** Permiten indicar dónde empieza y termina un elemento.

**grid-area** Puede usarse tanto para áreas nombradas como para especificar posiciones con líneas.

**Solapamiento** Dos o más ítems pueden ocupar la misma celda; el orden se controla con `z-index`.

### Propiedades útiles

**gap** Define el espacio entre filas y columnas.

**justify-items, align-items** Controlan la alineación de los ítems dentro de sus celdas.

**justify-content, align-content** Ajustan cómo se distribuye toda la cuadrícula dentro del contenedor.

**auto-fit, auto-fill** Facilitan la creación de cuadrículas responsivas, añadiendo o ajustando columnas automáticamente.

## Ventajas

**Potencia** Permite crear layouts complejos de forma declarativa y legible.

**Responsividad** Facilita la adaptación automática a distintos tamaños de pantalla.

**Flexibilidad** Posibilidad de solapar elementos y definir áreas de tamaño dinámico.

**Limitaciones** Requiere aprender nuevas propiedades y mentalidad diferente frente al flujo normal.

Puede ser excesivo para estructuras simples que se resuelven mejor con Flexbox (1.4).

```
1  .grid {
2    display: grid;
3    grid-template-columns: repeat(3, 1fr);
4    grid-template-rows: auto;
5    gap: 10px;
6  }
7
8  .grid div:nth-child(1) {
9    grid-column: 1 / 3; /* abarca dos columnas */
10 }
11
12 .grid div:nth-child(2) {
13   grid-row: 2 / 4; /* abarca dos filas */
14 }
```

```
1  .grid {
2    display: grid;
3    grid-template-areas:
4      "header header"
5      "sidebar main"
6      "footer footer";
7    grid-template-columns: 200px 1fr;
8    grid-template-rows: auto 1fr auto;
9  }
10
11 header { grid-area: header; }
12 aside  { grid-area: sidebar; }
13 main   { grid-area: main; }
14 footer { grid-area: footer; }
```

## 1.6. Multi-column Layout

**Concepto** El módulo de diseño multicolumna de CSS permite dividir el contenido de un contenedor en varias columnas, de manera similar al diseño de periódicos o revistas. El navegador reparte automáticamente el flujo de texto entre las columnas, ajustando su altura de forma equilibrada.

### Propiedades principales

**column-count** Define el número de columnas en que se divide el contenido.

**column-width** Establece el ancho ideal de cada columna; el navegador calcula cuántas entran en el espacio disponible.

**columns** Shorthand que combina **column-count** y **column-width**.

**column-gap** Define el espacio entre columnas.

**column-rule** Dibuja una línea divisoria entre columnas (similar a **border**, pero solo en el espacio entre columnas).

**column-span** Permite a un elemento ocupar todas las columnas (ejemplo: un título que atraviesa el ancho completo).

**break-inside** Controla saltos de columna, útil para evitar que un bloque se corte.

## Ventajas

**Rápido** Muy sencillo para crear maquetaciones tipo periódico.

**Automático** El navegador reparte el flujo del texto sin necesidad de calcular manualmente.

## Limitaciones

**Poco control** Difícil colocar elementos de forma precisa en columnas específicas.

**Desigualdad** Dependiendo del contenido, las columnas pueden quedar con alturas distintas.

**Flexibilidad** Menos versátil que Grid o Flexbox para layouts complejos.

## Casos de uso

**Artículos largos** Textos que se leen mejor en columnas estrechas.

**Revistas digitales** Diseños inspirados en periódicos y newsletters.

**Listados largos** Mostrar listas distribuidas automáticamente en varias columnas.

```
1 .texto {
2   column-count: 3;
3   column-gap: 20px;
4   column-rule: 1px solid #ccc;
5 }
```

Este código divide el texto en tres columnas, con separación de 20px y una línea divisoria gris.

```
1 .articulo {
2   column-width: 200px;
3   column-gap: 1.5em;
4 }
```

Aquí el navegador intentará colocar tantas columnas de 200px como quepan en el ancho disponible, respetando el espacio de 1.5em entre ellas.

## 1.7. CSS Table Layout

**Concepto** CSS permite emular el comportamiento de tablas HTML en cualquier elemento, utilizando las propiedades **display: table**, **display: table-row** y **display: table-cell**. Esto convierte bloques en estructuras tabulares que se comportan como una tabla tradicional, pero con mayor flexibilidad semántica (sin necesidad de usar la etiqueta **<table>**).

**Uso histórico** Antes de la llegada de Flexbox (1.4) y Grid (1.5), era común emplear el modelo de tabla para construir layouts completos de páginas web. Se creaban cabeceras, menús laterales y contenidos principales usando contenedores con **display: table** y celdas anidadas.

### Propiedades clave

**display: table** Define el contenedor principal que actúa como tabla.

**display: table-row** Representa una fila dentro de la tabla.

**display: table-cell** Crea celdas que se comportan como `<td>`, con alineación vertical y reparto de espacio.

**table-layout** Controla el algoritmo de distribución de celdas:

- auto** (por defecto) ajusta el ancho de las columnas según su contenido.
- fixed** calcula anchos en función de la primera fila, más eficiente para tablas grandes.

**border-collapse** Define cómo se combinan los bordes entre celdas (colapsados o separados).

## Ventajas

**Compatibilidad** Soportado en todos los navegadores desde hace mucho tiempo.

**Consistencia** Permite igualar alturas de columnas automáticamente, algo que antes era difícil de lograr.

## Limitaciones

**Semántica** Usar tablas para maquetación rompe con el significado estructural del HTML.

**Rigidez** Menos flexible para adaptarse a diseños responsivos modernos.

**Reemplazo** Flexbox y Grid ofrecen soluciones más potentes y adaptables.

## Casos actuales

**Datos tabulares** Representación de tablas de verdad, donde la semántica sí corresponde.

**Compatibilidad heredada** Código antiguo que todavía se mantiene en proyectos legacy.

**Casos puntuales** Cuando se necesita igualar la altura de celdas sin recurrir a técnicas modernas.

```

1  .contenedor {
2      display: table;
3      width: 100%;
4      border-collapse: collapse;
5  }
6
7  .fila {
8      display: table-row;
9  }
10
11 .celda {
12     display: table-cell;
13     padding: 10px;
14     border: 1px solid #ccc;
15 }

```

Este ejemplo genera una estructura similar a una tabla con CSS, sin necesidad de usar `<table>`.

## 1.8. Exclusions y Shapes

**Concepto** Las propiedades de Exclusions y Shapes permiten que el contenido (normalmente texto) fluya alrededor de formas no rectangulares. Se emplea principalmente la propiedad **shape-outside**, aplicada sobre un elemento flotante, para definir el contorno alrededor del cual se ajusta el texto.

### Propiedades principales

**shape-outside** Define la forma alrededor de la cual fluye el texto. Puede recibir valores como `circle()`, `ellipse()`, `polygon()`, o usar la transparencia de una imagen.

**clip-path** Aunque no pertenece al mismo módulo, a menudo se combina para recortar la parte visible de un elemento, mientras que **shape-outside** controla cómo fluye el texto.

**float** Es necesario para que el navegador aplique la forma definida con **shape-outside**, ya que el texto solo fluye alrededor de elementos flotantes.

### Ventajas

**Creatividad** Permite diseños editoriales más atractivos y no rectangulares.

**Control preciso** Se puede definir exactamente cómo fluye el texto alrededor de una forma.

### Limitaciones

**Compatibilidad** Soporte limitado en navegadores; no todas las funciones están implementadas de forma uniforme.

**Dependencia de floats** Solo funciona en elementos flotados, lo cual limita su uso en layouts modernos.

**Accesibilidad** Usar formas complejas puede dificultar la lectura si no se aplica con cuidado.

### Casos de uso

**Diseños editoriales** Revistas digitales, newsletters, artículos con imágenes llamativas.

**Diseño experimental** Interfaces creativas con elementos gráficos integrados en el texto.

**Ejercicios académicos** Para mostrar el potencial de CSS más allá de las cajas rectangulares.

```
1  .circulo {
2      float: left;
3      width: 200px;
4      height: 200px;
5      shape-outside: circle(50%);
6      clip-path: circle(50%);
7      background: url(imagen.jpg) no-repeat center/cover;
8      margin: 20px;
9  }
```

Este código crea una imagen circular a la izquierda y el texto fluye alrededor siguiendo el contorno definido.

```
1  .poligono {
2      float: right;
3      width: 250px;
4      height: 250px;
5      shape-outside: polygon(0 0, 100% 0, 80% 100%, 20% 100%);
6      clip-path: polygon(0 0, 100% 0, 80% 100%, 20% 100%);
7      background: lightblue;
8      margin: 20px;
9  }
```

Aquí el texto se ajusta a un polígono trapezoidal definido por coordenadas.

## 2. Media Queries en CSS

**Concepto** Las **media queries** son una característica de CSS que permiten aplicar estilos condicionales según las características del dispositivo o del entorno donde se muestra la página. Se utilizan principalmente para lograr diseños **responsivos**, adaptando el layout a distintos anchos de pantalla, orientaciones o preferencias del usuario.

**Sintaxis básica** Una media query combina la regla `@media` con una condición lógica:

```
1  @media (condición) {  
2      /* Estilos aplicados solo si se cumple la condición */  
3  }
```

### Tipos de condiciones

**Ancho y alto de la ventana** `min-width`,  
`max-width`, `min-height`, `max-height`.

**Orientación** `orientation: portrait | landscape`.

**Resolución y densidad de píxeles** `min-resolution`, `max-resolution`, útil para pantallas retina.

**Modo de color** `prefers-color-scheme: light | dark`.

**Movimiento reducido** `prefers-reduced-motion: reduce`.

**Tipos de dispositivo** (obsoletos en parte): `screen`, `print`, etc.

**Uso en diseño responsivo Mobile First** Se escriben estilos base para móviles y se agregan media queries con `min-width` para pantallas más grandes.

**Desktop First** Se escriben estilos para pantallas grandes y se ajustan con `max-width` para móviles.

**Ventajas Adaptabilidad** Permiten que un mismo sitio se vea bien en móviles, tablets, portátiles y pantallas grandes.

**Accesibilidad** Respetan preferencias del usuario como modo oscuro o reducción de movimiento.

**Precisión** Posibilitan cambios puntuales en la interfaz según contexto y dispositivo.

```
1  /* Ejemplo 1: estilos para pantallas pequeñas */  
2  @media (max-width: 600px) {  
3      body { font-size: 14px; }  
4  }  
5  
6  /* Ejemplo 2: estilos para modo oscuro */  
7  @media (prefers-color-scheme: dark) {  
8      body { background: #111; color: #eee; }  
9  }  
10  
11 /* Ejemplo 3: orientación horizontal */  
12 @media (orientation: landscape) {  
13     .menu { display: flex; }  
14 }
```

sectionUnidades modernas y fluidas

**Viewport units** `vw`, `vh`, `vmin`, `vmax`. Ejemplo:

```
1  h1 { font-size: 10vw; } /* Escala con ancho de ventana */
```

**Funciones matemáticas** `clamp()`, `min()`, `max()`:

```
1  h1 { font-size: clamp(2rem, 5vw, 4rem); }
```

### 3. Variables CSS (Custom Properties)

**Concepto** Las variables CSS (o *custom properties*) permiten definir valores reutilizables en las hojas de estilo. Se declaran mediante la sintaxis **-nombre** y se accede a ellas con la función **var(-nombre)**. A diferencia de las variables de preprocesadores (Sass, Less), estas existen en tiempo de ejecución y heredan según el árbol del DOM.

```
1 :root {
2   --color-primario: #2D6CDF;
3   --espaciado: 1rem;
4 }
5 a {
6   color: var(--color-primario);
7   margin: var(--espaciado);
8 }
```

#### Ventajas

- Permiten mantener un diseño consistente y facilitan la refactorización.
  - Soportan herencia: los valores cambian según el contexto del elemento.
  - Se pueden actualizar dinámicamente con JavaScript, habilitando temas o personalización en tiempo real.

**Temas claro/oscuro** Una de las aplicaciones más comunes es la definición de temas (p. ej. claro y oscuro). Basta con redefinir las variables en un selector diferente, como en el siguiente ejemplo:

```
1 body.light {
2   --color-primario: #2D6CDF;
3   --bg: #fff;
4   --fg: #000;
5 }
6
7 body.dark {
8   --color-primario: #8ab4f8;
9   --bg: #111;
10  --fg: #eee;
11 }
12
13 body {
14   background: var(--bg);
15   color: var(--fg);
16 }
```

**Uso dinámico con JavaScript** Además, es posible modificar variables desde JavaScript, lo que permite personalización del usuario:

```
1 document.documentElement.style
2   .setProperty('--color-primario', '#FF5733');
```

## 4. Accesibilidad y UX

La accesibilidad en la web no es solamente una obligación legal en muchos contextos, sino una condición necesaria para ofrecer una buena experiencia de usuario (UX). Una página accesible permite que todas las personas, incluidas aquellas con discapacidades visuales, auditivas, motoras o cognitivas, puedan interactuar con los contenidos de manera adecuada. En esta sección se detallan algunos principios fundamentales y buenas prácticas que todo proyecto web debe considerar.

### 4.1. Contraste y legibilidad

El contraste entre el texto y el fondo es uno de los aspectos más relevantes en la accesibilidad. Las pautas WCAG (*Web Content Accessibility Guidelines*) establecen unos mínimos recomendados:

- Para texto normal: una relación de contraste mínima de **4.5:1**.
- Para texto grande (mayor a 18pt o 14pt en negrita): mínimo de **3:1**.

## Relación de contraste en accesibilidad

La **relación de contraste** mide la diferencia de luminosidad entre el color del texto y el color de fondo. Según las WCAG 2.1, el contraste mínimo recomendado para *texto normal* es de **4.5:1**.

### Definición matemática

Dada la *luminancia relativa* de dos colores  $L_1$  y  $L_2$ , donde  $L_1$  es el valor más claro y  $L_2$  el más oscuro, la relación de contraste se calcula como:

$$\text{Contraste} = \frac{L_1 + 0,05}{L_2 + 0,05}$$

La luminancia relativa  $L$  de un color en formato RGB normalizado (valores entre 0 y 1) se obtiene aplicando una corrección gamma:

$$R_s = \begin{cases} \frac{R}{12,92}, & \text{si } R \leq 0,03928 \\ \left( \frac{R + 0,055}{1,055} \right)^{2,4}, & \text{si } R > 0,03928 \end{cases}$$
$$G_s = \dots \quad B_s = \dots$$

$$L = 0,2126 \cdot R_s + 0,7152 \cdot G_s + 0,0722 \cdot B_s$$

donde  $R$ ,  $G$ ,  $B$  son los valores normalizados de cada canal de color.

### Ejemplos

- **Negro sobre blanco:**  $L_{\text{blanco}} = 1,0$ ,  $L_{\text{negro}} = 0,0$

$$\text{Contraste} = \frac{1,0 + 0,05}{0,0 + 0,05} = 21 : 1$$

Máximo contraste posible.

- **Gris #777777 sobre blanco:**  $L_{\text{gris}} \approx 0,184$ ,  $L_{\text{blanco}} = 1,0$

$$\text{Contraste} = \frac{1,0 + 0,05}{0,184 + 0,05} \approx 4,48 : 1$$

Apenas cumple el mínimo recomendado para texto normal.

- **Gris #AAAAAA sobre blanco:**  $L_{gris} \approx 0,402$ ,  $L_{blanco} = 1,0$

$$Contraste = \frac{1,0 + 0,05}{0,402 + 0,05} \approx 2,3 : 1$$

No cumple el requisito de accesibilidad.

Un contraste mínimo de **4.5:1** asegura que el texto sea legible para la mayoría de las personas, incluyendo aquellas con visión reducida. Para texto grande (18pt o 14pt en negrita), el mínimo aceptado es **3:1**, mientras que el nivel AAA exige hasta **7:1**.

Además del contraste de color, es importante no depender únicamente del color para transmitir información. Por ejemplo, los enlaces deben subrayarse o diferenciarse con un estilo visual claro, no solo mediante el color.

```
1 /* Ejemplo de mal contraste */
2 .texto-poco-visible {
3   color: #999;
4   background: #aaa;
5 }
6
7 /* Ejemplo accesible */
8 .texto-accesible {
9   color: #000;
10  background: #fff;
11 }
```

## 4.2. Preferencias del usuario

El diseño inclusivo reconoce que los usuarios pueden tener configuraciones específicas en su sistema operativo o navegador. Entre estas preferencias destacan:

- **Reducción de movimiento:** usuarios con mareos o epilepsia fotosensible.
- **Modo oscuro o claro:** preferencia estética o de descanso visual.
- **Tamaño de fuente preferido:** configuraciones de accesibilidad en el sistema.

CSS permite detectar estas preferencias y adaptar la interfaz de manera automática:

```
1 @media (prefers-reduced-motion: reduce) {
2   * { animation: none !important; }
3 }
4
5 @media (prefers-color-scheme: dark) {
6   body {
7     background: #111;
8     color: #eee;
9   }
10 }
```

## 4.3. Uso de etiquetas semánticas

El HTML semántico es clave para la accesibilidad. Los lectores de pantalla y las tecnologías asistivas dependen de la estructura correcta del documento. Algunas recomendaciones:

- Utilizar `<header>`, `<main>`, `<footer>` en lugar de `<div>`.
- Emplear encabezados jerárquicos (`<h1>` a `<h6>`) de forma ordenada.

- Incluir descripciones en imágenes mediante `alt`.
- Asegurar que los formularios tengan etiquetas (`<label>`) asociadas a los campos.

```
1 <form>
2   <label for="email">Correo electrónico:</label>
3   <input type="email" id="email" name="email">
4 </form>
```

#### 4.4. Navegación por teclado

Un principio fundamental es que todo contenido y funcionalidad debe poder usarse sin ratón. Esto implica:

- Asegurar que los enlaces y botones sean alcanzables con **Tab**.
- Utilizar `:focus` en CSS para resaltar el elemento activo.
- Evitar secuestro del foco con scripts.

```
1 button:focus,
2 a:focus {
3   outline: 3px solid #2D6CDF;
4   outline-offset: 2px;
5 }
```

#### 4.5. Compatibilidad con lectores de pantalla

Los lectores de pantalla traducen el contenido web en voz o braille. Para optimizar la experiencia:

- Usar atributos `aria-` para comunicar estados dinámicos.
- Evitar que elementos visuales carezcan de descripción.
- Utilizar roles (`role="alert"`, `role="dialog"`, etc.) donde sea necesario.

```
1 <button aria-pressed="false">Activar</button>
2 <div role="alert">Su sesión ha expirado</div>
```

#### 4.6. Diseño responsivo e inclusivo

El diseño responsivo no solo se refiere al tamaño de pantalla, sino también a la adaptabilidad para diferentes capacidades. Algunos puntos clave:

- Diseñar interfaces que funcionen tanto en pantallas táctiles como con teclado.
- Permitir zoom sin romper el diseño (mínimo 200 % recomendado).
- Mantener tamaños de toque adecuados (botones grandes y espaciados).

## 4.7. Experiencia de usuario (UX)

La accesibilidad impacta directamente en la UX: un sistema usable para todos es un sistema más robusto. Un buen diseño accesible suele coincidir con un diseño intuitivo y claro para cualquier usuario.

- Formularios claros y con mensajes de error comprensibles.
- Botones con etiquetas que expliquen la acción.
- Jerarquía visual coherente y predecible.
- Textos comprensibles, evitando tecnicismos innecesarios.

## 4.8. Conclusión

La accesibilidad no debe considerarse una etapa final, sino un principio transversal en el diseño y desarrollo. Aplicando buenas prácticas como contraste suficiente, semántica correcta, soporte a preferencias del usuario y navegación por teclado, no solo se cumple con estándares, sino que también se mejora la experiencia de toda la audiencia.

## 5. Responsive avanzado

El diseño *responsive* moderno no se limita únicamente a usar **media queries** tradicionales, sino que incorpora nuevas técnicas y conceptos que permiten interfaces más adaptables, modulares y escalables. En esta sección se abordan dos enfoques clave: la estrategia *Mobile First* y el uso de *Container Queries*.

### 5.1. Estrategia Mobile First

El enfoque **Mobile First** consiste en diseñar y aplicar los estilos base pensando primero en dispositivos móviles. Posteriormente, se amplían las reglas para pantallas más grandes mediante `@media (min-width)`.

- Garantiza una buena experiencia en pantallas pequeñas, que suelen ser las más utilizadas.
- Obliga a priorizar contenido y simplificar la interfaz.
- Reduce la carga inicial de CSS en móviles, pues las reglas adicionales solo se aplican en pantallas grandes.

```
1 /* Estilos base: móviles */
2 body {
3   font-size: 1rem;
4   padding: 1rem;
5 }
6
7 nav ul {
8   display: flex;
9   flex-direction: column;
10 }
11
12 /* A partir de tablets/escritorio */
13 @media (min-width: 768px) {
14   nav ul {
15     flex-direction: row;
16     gap: 2rem;
17   }
18 }
```

Este patrón garantiza que el sitio sea usable en cualquier contexto, comenzando desde la restricción más fuerte (pantallas pequeñas) y expandiendo hacia configuraciones más amplias.

### 5.2. Container Queries

Las **Container Queries** son una de las innovaciones más recientes en CSS. Mientras que las **media queries** dependen del ancho de la ventana (*viewport*), las container queries aplican estilos en función del tamaño del contenedor padre. Esto permite crear componentes realmente independientes y reutilizables.

```
1 /* Definir un contenedor */
2 .card-container {
3   container-type: inline-size;
4 }
5
6 /* Ajustar estilos según el tamaño del contenedor */
7 @container (min-width: 400px) {
8   .card {
9     flex-direction: row;
```

```
10 align-items: center;
11 }
12 }
```

Ventajas principales:

- Un componente se adapta al espacio disponible, no a la pantalla completa.
- Se facilita la construcción de librerías de UI independientes.
- Menor dependencia del diseño global: cada bloque puede ser responsivo por sí mismo.

### 5.3. Unidades relativas y modernas

Para un diseño verdaderamente flexible es importante usar unidades relativas:

- `em` y `rem` para tipografía escalable.
- `vw` y `vh` para ocupar porcentajes del viewport.
- `min()`, `max()` y `clamp()` para definir rangos fluidos.

```
1 h1 {
2   font-size: clamp(1.5rem, 5vw, 3rem);
3 }
```

Este ejemplo garantiza que el título nunca será más pequeño que 1.5rem ni más grande que 3rem, pero crecerá proporcionalmente con el ancho de la pantalla dentro de ese rango.

### 5.4. Grid Layout responsivo

El sistema de CSS Grid permite crear diseños adaptables sin necesidad de demasiadas media queries. El uso de funciones como `minmax()` y `auto-fit` hace posible construir rejillas fluidas.

```
1 .grid {
2   display: grid;
3   grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
4   gap: 1rem;
5 }
```

En este ejemplo, la rejilla se ajusta automáticamente mostrando tantas columnas como quepan, siempre respetando un ancho mínimo de 250px por tarjeta.

### 5.5. Patrones avanzados de responsive

Algunas técnicas adicionales que enriquecen el diseño adaptativo son:

- **Responsive Typography:** escalado fluido de fuentes con `clamp()`.
- **Responsive Images:** uso de `srcset` y `sizes` para servir imágenes optimizadas según el dispositivo.
- **Aspect Ratio:** asegurar proporciones constantes en elementos multimedia con `aspect-ratio`.

```
1 img.responsive {
2   width: 100%;
3   height: auto;
4   aspect-ratio: 16 / 9;
5 }
```

## 5.6. Conclusión

El diseño responsivo ha evolucionado desde simples **media queries** hacia enfoques más modulares, adaptativos y potentes. Aplicar *Mobile First*, combinarlo con *Container Queries*, usar unidades relativas y aprovechar CSS Grid y Flexbox permite construir interfaces flexibles y modernas. El objetivo final es que la experiencia de usuario sea óptima en cualquier dispositivo, tamaño de pantalla o contexto de uso.

## 6. Calidad y mantenibilidad

La calidad del código CSS no solo se mide por su resultado visual, sino también por su capacidad de mantenerse y evolucionar en proyectos grandes y de larga duración. Un CSS mal estructurado tiende a crecer de manera desordenada, con duplicados, reglas contradictorias y dificultad para escalar. Para evitar estos problemas, existen metodologías y buenas prácticas que ayudan a mantener el orden y la coherencia.

### 6.1. Metodologías de organización

A lo largo de los años se han desarrollado diferentes enfoques para estructurar hojas de estilo. Entre los más destacados se encuentran:

#### BEM (Bloque, Elemento, Modificador)

El sistema BEM (*Block, Element, Modifier*) propone una convención de nombres clara y predecible. Se divide en tres niveles:

- **Bloque:** componente independiente (`.menu`).
- **Elemento:** parte interna de un bloque (`.menu__item`).
- **Modificador:** variación de un bloque o elemento (`.menu__item-activo`).

```
1 .menu { }  
2 .menu__item { }  
3 .menu__item--activo { font-weight: bold; }
```

Ventajas de BEM:

- Facilita la lectura y el mantenimiento del código.
- Reduce la posibilidad de conflictos de estilos.
- Promueve la reutilización de componentes.

#### ITCSS (Inverted Triangle CSS)

El enfoque ITCSS propone organizar las hojas de estilo en capas, desde lo más general hasta lo más específico. La estructura suele ser:

1. **Settings:** variables globales, colores, tipografías.
2. **Tools:** funciones y mixins.
3. **Generic:** estilos base, resets.
4. **Elements:** estilos de HTML puro (`h1`, `a`, etc.).
5. **Objects:** patrones reutilizables (`.container`, `.grid`).
6. **Components:** bloques específicos de UI (`.card`, `.navbar`).
7. **Utilities:** clases de ayuda (`.text-center`, `.hidden`).

Este modelo facilita la escalabilidad en proyectos grandes y asegura que los estilos más específicos no se mezclen con los genéricos.

## SMACSS (Scalable and Modular Architecture for CSS)

SMACSS organiza el CSS en cinco categorías:

1. **Base:** estilos predeterminados para elementos HTML.
2. **Layout:** estructura de la página.
3. **Module:** componentes reutilizables.
4. **State:** estados o variaciones de módulos.
5. **Theme:** temas o personalizaciones globales.

Su objetivo es dividir el CSS en piezas pequeñas y coherentes, que se puedan mantener y reutilizar fácilmente.

### 6.2. Buenas prácticas en CSS

Más allá de las metodologías, existen recomendaciones generales que mejoran la calidad del código:

#### Uso de variables

Definir variables CSS para colores, tamaños y tipografías permite mantener consistencia y reducir errores.

```
1 :root {
2   --color-primario: #2D6CDF;
3   --espaciado: 1rem;
4 }
5
6 button {
7   background: var(--color-primario);
8   margin: var(--espaciado);
9 }
```

#### Evitar duplicados

Un código con reglas repetidas es más difícil de mantener y aumenta el peso del CSS. Se recomienda aplicar el principio DRY (*Don't Repeat Yourself*) para centralizar estilos.

```
1 /* Ejemplo incorrecto */
2 .card { margin: 1rem; }
3 .box { margin: 1rem; }
4
5 /* Ejemplo optimizado */
6 .spaced { margin: 1rem; }
7 .card { @extend .spaced; }
8 .box { @extend .spaced; }
```

#### Comentarios claros y útiles

Los comentarios deben servir para explicar decisiones o secciones importantes del código, no repetir lo obvio.

```
1 /* Botones principales: usan color corporativo */
2 .btn-primario {
3     background: var(--color-primario);
4     color: #fff;
5 }
```

### Consistencia en formato y estilo

Mantener un estilo uniforme de escritura evita confusión:

- Indentación coherente (2 o 4 espacios).
- Orden alfabético de propiedades o agrupación lógica.
- Uso consistente de comillas, minúsculas y unidades.

### Modularidad y reutilización

Crear clases reutilizables y componentes independientes reduce la complejidad y favorece el mantenimiento. Se recomienda dividir el CSS en varios archivos organizados por módulos, especialmente en proyectos grandes.

### 6.3. Conclusión

La calidad y mantenibilidad del CSS dependen de aplicar metodologías claras como BEM, ITCSS o SMACSS, junto con buenas prácticas de organización, documentación y consistencia. Un CSS bien estructurado no solo mejora la productividad del equipo, sino que también garantiza que el proyecto pueda crecer y evolucionar sin perder coherencia ni generar deuda técnica.

## 7. Tendencias actuales

El ecosistema de CSS se encuentra en constante evolución. En los últimos años se han incorporado nuevas funcionalidades al lenguaje que permiten diseñar interfaces más potentes, flexibles y adaptables sin depender tanto de preprocesadores o hacks. Entre las tendencias más relevantes se encuentran el uso de **Subgrid**, la **anidación nativa (Nesting)** y las **propiedades lógicas (Logical Properties)**. Estas características no solo simplifican el código, sino que también promueven una web más accesible y adaptable a distintos contextos culturales y tecnológicos.

### 7.1. Subgrid

El modelo de CSS Grid revolucionó la maquetación en la web al permitir dividir el espacio en rejillas bidimensionales. Sin embargo, hasta hace poco existía una limitación: los elementos hijos no podían heredar directamente la definición de columnas o filas del contenedor padre.

Con la propiedad **subgrid**, es posible que un grid hijo utilice los mismos *tracks* definidos en el grid padre, garantizando alineaciones perfectas.

```
1 /* Definición del grid padre */
2 .container {
3   display: grid;
4   grid-template-columns: 1fr 2fr 1fr;
5   gap: 1rem;
6 }
7
8 /* El grid hijo hereda las columnas del padre */
9 .item {
10  display: grid;
11  grid-template-columns: subgrid;
12  grid-column: span 3;
13 }
```

Ventajas principales:

- Se simplifica la alineación entre elementos anidados.
- Se reduce la necesidad de repetir definiciones de columnas o filas.
- Facilita diseños editoriales o de tablas complejas.

Aplicación práctica: en maquetaciones de periódicos digitales o dashboards, donde múltiples bloques de contenido deben alinearse verticalmente en columnas compartidas.

### 7.2. Nesting en CSS

Durante mucho tiempo, los desarrolladores dependieron de preprocesadores como Sass o Less para escribir reglas anidadas. Hoy en día, **CSS soporta anidación nativa**, lo que permite agrupar estilos relacionados de manera más lógica y legible.

```
1 /* Antes (Sass) */
2 .card {
3   border: 1px solid #ccc;
4   .title {
5     font-weight: bold;
6   }
7   .content {
8     font-size: 0.9rem;
9   }
10 }
11
```

```

12 /* Ahora en CSS puro */
13 .card {
14     border: 1px solid #ccc;
15
16     & .title {
17         font-weight: bold;
18     }
19
20     & .content {
21         font-size: 0.9rem;
22     }
23 }

```

Beneficios:

- Reducción del número de selectores redundantes.
- Código más compacto y fácil de mantener.
- Integración nativa en navegadores modernos, sin necesidad de compilación.

Limitaciones actuales:

- No todos los navegadores soportan el 100 % de la sintaxis, por lo que a veces se requiere un *fallback*.
- Es importante no abusar de la anidación profunda, ya que genera CSS muy específico y difícil de sobrescribir.

### 7.3. Logical Properties

Las **propiedades lógicas** representan un cambio de paradigma en CSS. Tradicionalmente, las propiedades como `margin-left`, `padding-top` o `border-right` estaban atadas a la dirección física del contenido. Esto generaba problemas en interfaces que debían adaptarse a distintos sistemas de escritura, como el **LTR (Left-to-Right)** o el **RTL (Right-to-Left)**.

Ejemplo clásico: un margen definido en `left` no tiene sentido si el idioma cambia a árabe o hebreo (RTL). Con las propiedades lógicas, el código se abstrae de la dirección del texto:

```

1 /* Propiedades tradicionales */
2 .card {
3     margin-left: 2rem;
4     padding-top: 1rem;
5 }
6
7 /* Equivalente con propiedades lógicas */
8 .card {
9     margin-inline-start: 2rem;
10    padding-block-start: 1rem;
11 }

```

Ventajas:

- Interfaces automáticamente adaptables a distintos idiomas.
- Consistencia en el diseño sin duplicar reglas.
- Preparación para un entorno globalizado y accesible.

Ejemplo avanzado con `margin-inline` y `padding-block`:

```

1 .button {
2   margin-inline: 1rem; /* Aplica a ambos lados en el eje inline */
3   padding-block: 0.5rem; /* Aplica arriba y abajo según escritura */
4 }

```

## 7.4. Otras tendencias emergentes

Además de las tres mencionadas, existen otras características modernas en CSS que complementan estas tendencias:

### Funciones modernas

- `clamp()`: define valores mínimos, preferidos y máximos.
- `min()` y `max()`: cálculos dinámicos.
- `calc()`: operaciones matemáticas flexibles.

```

1 h1 {
2   font-size: clamp(1.5rem, 5vw, 3rem);
3 }

```

### Container Queries + Subgrid

La combinación de *container queries* con *subgrid* permite componentes totalmente autónomos y flexibles, ajustándose no solo al viewport, sino al espacio disponible en su contenedor.

### Variables de nivel de vista (Viewport Units de nueva generación)

Unidades como `lvh`, `svh` y `dvh` resuelven problemas con las barras de navegación móviles, asegurando medidas más estables.

### Soporte a prefers-queries

Además de `prefers-reduced-motion` y `prefers-color-scheme`, están surgiendo nuevas consultas de medios para accesibilidad, como `prefers-contrast`, que permiten interfaces todavía más inclusivas.

## 7.5. Impacto en la práctica profesional

Estas tendencias reflejan un cambio importante en el rol del CSS:

- Deja de ser un lenguaje complementario para convertirse en una capa central del desarrollo web.
- Reduce la necesidad de preprocesadores o utilidades externas.
- Permite construir sistemas de diseño modulares, internacionales y accesibles.

Ejemplo de integración práctica:

```

1 .article {
2   display: grid;
3   grid-template-columns: subgrid;
4   container-type: inline-size;
5   padding-inline: 2rem;

```

```
6
7  & h2 {
8    font-size: clamp(1.2rem, 3vw, 2rem);
9  }
10
11  & p {
12    line-height: 1.6;
13  }
14 }
```

## 7.6. Conclusión

El futuro del CSS se centra en **flexibilidad**, **modularidad** y **accesibilidad**. Con *subgrid*, los diseños complejos se simplifican. Con *nesting*, la sintaxis se vuelve más clara y cercana a los preprocesadores. Con *logical properties*, el CSS se prepara para un mundo multilingüe y diverso.

Estas tendencias confirman que el CSS es un lenguaje vivo, en constante evolución, que busca adaptarse a los retos de la web moderna y que, con cada nueva especificación, acerca al desarrollador a la creación de interfaces más robustas, escalables y universales.

## 8. Ejercicios con solución

### 8.1. Ejercicio Especificidad y colisión de estilos

Dado el siguiente HTML y CSS, determina el color final del texto del párrafo y justifica por especificidad:

```
1 <body>
2   <p id="aviso" class="destacado nota">Hola CSS</p>
3 </body>
4
5 <style>
6   p           { color: black; }
7   .destacado  { color: green; }
8   p.nota      { color: orange; }
9   #aviso      { color: blue; }
10  p#aviso     { color: purple; }
11 </style>
```

**Solución.** Especificidad (aprox.):  $p=(0,0,0,1)$ ;  $.destacado=(0,0,1,0)$ ;  $p.nota=(0,0,1,1)$ ;  $\#aviso=(0,1,0,0)$ ;  $p\#aviso=(0,1,0,1)$ . Gana  $p\#aviso$  por mayor especificidad  $\Rightarrow$  color **morado**. (En el temario: reglas de selectores y colisión por especificidad/orden).

### 8.2. Ejercicio Selectores combinados y herencia

Estiliza sólo los **span** dentro de **a** que estén a su vez dentro de un **p** con clase **nota**. Deben ser *rojos*, subrayados y no afectar a otros **span**.

**Solución.**

```
1 p.nota a span {
2   color: red;
3   text-decoration: underline;
4 }
```

El combinador descendiente y la clase concreta limitan el alcance. (Temario: combinaciones de selectores y descendencia).

### 8.3. Ejercicio Modelo de caja y box-sizing

Crea dos cajas de 300px de ancho visual idéntico: una usando el box model clásico y otra con `box-sizing: border-box`. Cada una debe mostrar borde de 10px y `padding` de 20px. Explica la diferencia. (1.1) **Solución.**

```
1 <div class="clasico">Clásico</div>
2 <div class="borderbox">Border-box</div>
3
4 <style>
5 div { border:10px solid #333; padding:20px; margin:8px 0; background:#
6   f7f7f7; }
7 .clasico { width:300px; /* ancho contenido: +padding+borde
8   => mayor total */ }
9 .borderbox { width:300px; box-sizing:border-box; /* total fijo a 300px */
10 }
11 </style>
```

Con el modelo clásico, el ancho total = `width` + `padding` + `border`. Con `border-box`, el total se mantiene en 300px. (Temario: modelo de caja (1.1, márgenes, padding, borde).

## 8.4. Ejercicio Unidades y accesibilidad tipográfica

Define tipografías escalables: tamaño base en 16px en `html`, párrafos a 1rem, titulares H1 a `clamp(1.8rem, 4vw, 3rem)`. Justifica el uso de `rem` y `vw`.

**Solución.**

```
1 html { font-size:16px; }
2 p   { font-size:1rem; line-height:1.6; }
3 h1   { font-size:clamp(1.8rem, 4vw, 3rem); }
```

`rem` hereda de raíz (consistente con preferencias del usuario); `vw` permite que el título escale con el ancho de la ventana. (Temario: unidades absolutas/relativas y porcentajes).

## 8.5. Ejercicio Paleta y contraste

Define una paleta con variables CSS y aplica contraste adecuado para texto sobre fondo. Usa dos temas (claro/oscuro) con una clase en `body`.

### Teoría esencial

- **Variables CSS (“custom properties”)**: se definen con `-nombre` (p. ej., `-bg`) y se consumen con `var(-nombre)`. Son heredables y se ven afectadas por la cascada; esto permite *tematizar* una interfaz cambiando sólo los valores en un ámbito (por ejemplo, en `body`).
- **Temas claro/oscuro**: basta con sobrescribir las variables en un selector que sólo aplique cuando el documento esté en modo oscuro (p. ej., añadiendo la clase `dark` al `<body>`).
- **Contraste**: para legibilidad, usa combinaciones texto/fondo con buena relación de contraste (como guía, *normal* 4.5:1; *grande* 3:1). Las variables te permiten cambiar paletas sin romper el contraste si eliges colores adecuados para `-bg` y `-fg`.

### Paso a paso

**1) Define la paleta base en `:root`** Coloca los colores por defecto (tema claro) en el ámbito raíz:

```
/* Tema claro (por defecto) */
:root {
  --bg: #ffffff; /* fondo */
  --fg: #222222; /* texto principal */
  --prim:#2D6CDF; /* color de énfasis/enlaces */
}
```

**2) Crea el tema oscuro sobrescribiendo en `body.dark`** Cuando el documento tenga la clase `dark` en `<body>`, reasigna las variables:

```
/* Tema oscuro (activo cuando <body class="dark">) */
body.dark {
  --bg: #0f1115;
  --fg: #e8eaed;
  --prim:#8ab4f8;
}
```

**3) Aplica las variables a los elementos** Usa las variables en las propiedades que pintan el UI:

```
body {
  background: var(--bg);
  color:      var(--fg);
}

a {
  color: var(--prim);
  text-decoration: underline;
}
```

#### 4) Activa/cambia el tema

```
<!-- Claro por defecto -->
<body>
  ...
</body>

<!-- Oscuro: añade la clase -->
<body class="dark">
  ...
</body>
```

*Opcional (JS de ejemplo para alternar):*

```
document.querySelector('#toggle').addEventListener('click', () => {
  document.body.classList.toggle('dark');
});
```

**5) Verifica el contraste** Comprueba que `var(-fg)` sobre `var(-bg)` (y `var(-prim)` donde haya texto) mantienen buena legibilidad en ambos temas. Si algún par falla, ajusta ligeramente luminosidad/saturación de `-fg` o `-bg` hasta alcanzar un contraste adecuado.

#### Solución mínima completa (lista para copiar)

```
/* 1) Paleta por defecto */
:root {
  --bg: #ffffff;
  --fg: #222222;
  --prim:#2D6CDF;
}

/* 2) Sobrescritura en modo oscuro */
body.dark {
  --bg: #0f1115;
  --fg: #e8eaed;
  --prim:#8ab4f8;
}

/* 3) Aplicación */
body { background: var(--bg); color: var(--fg); }
a { color: var(--prim); text-decoration: underline; }
```

## Notas

- El enfoque usa **un único mapa de tokens** (variables) con **dos asignaciones** (claro/oscuro), lo que facilita el mantenimiento y la extensión (botones, tarjetas, bordes, etc.).
- Para ampliar, crea más variables (p.ej., `-muted`, `-border`, `-surface`) y aplícalas por componentes.

### 8.6. Ejercicio `display` vs `visibility` y flujo

Ocultar el tercer elemento de una cuadrícula de 9 bloques *sin* recolocar el resto; luego ocúltalo recolocando el flujo.

#### Teoría esencial

- **`visibility:hidden`**: el elemento sigue existiendo en el flujo del documento, ocupa su espacio reservado, pero su contenido no se renderiza. Ideal cuando queremos “ocultar a la vista” algo sin modificar la distribución general.
- **`display:none`**: el elemento se elimina del flujo visual. No ocupa espacio y los demás elementos se recolocan automáticamente como si no existiera.
- **Implicación práctica**: elegir uno u otro depende de si se desea mantener el hueco del elemento (ocultación sin recolocación) o si se quiere que la cuadrícula/rejilla se adapte (ocultación con recolocación).

#### Paso a paso

1) **Crear la cuadrícula de 9 elementos** Usamos CSS Grid para disponer los elementos en una matriz 3x3:

```
.grid {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  gap: 12px;
}
.item {
  background: #ddd;
  padding: 12px;
}
```

2) **Ocultar el tercer elemento sin recolocar el resto** Asignamos una clase al tercer elemento, usando `visibility:hidden`:

```
.item.hidden-space {
  visibility: hidden; /* sigue ocupando su espacio */
}
```

Resultado: la cuadrícula conserva la forma 3x3, pero el bloque queda vacío.

3) **Ocultar el tercer elemento recolocando el flujo** En este caso aplicamos `display:none`:

```
.item.hidden-flow {
  display: none; /* no ocupa sitio */
}
```

Resultado: la cuadrícula se reorganiza y pasa a mostrar 8 elementos compactos.

## Código completo

```
1 .grid {
2   display: grid;
3   grid-template-columns: repeat(3, 1fr);
4   gap: 12px;
5 }
6 .item {
7   background: #ddd;
8   padding: 12px;
9 }
10 .item.hidden-space { visibility: hidden; } /* ocupa sitio */
11 .item.hidden-flow { display: none; }      /* no ocupa sitio */
```

## Conclusión

- **visibility:hidden** es útil cuando queremos ocultar visualmente un elemento sin alterar la disposición general.
- **display:none** elimina el elemento del flujo, provocando que los demás se ajusten automáticamente.

Esto ilustra claramente la diferencia entre ocultar un elemento manteniendo su espacio y retirarlo por completo del flujo de maquetación .

## 8.7. Ejercicio Responsive con media queries

Haz que una barra de navegación horizontal pase a vertical bajo 640px.

### Teoría esencial

- **Flexbox**: facilita organizar elementos en fila o en columna, con control de alineación, espaciado y dirección.
- **Media queries**: permiten aplicar reglas de estilo condicionales según las características del dispositivo o ventana (por ejemplo, anchura del viewport).
- **Objetivo**: crear una barra de navegación que se muestre horizontal en pantallas grandes y que cambie a disposición vertical en pantallas pequeñas (menores a 640px de ancho).

### Paso a paso

**1) Definir la estructura HTML** Creamos una barra de navegación simple con enlaces:

```
<nav class="nav">
  <a>Inicio</a>
  <a>Blog</a>
  <a>Contacto</a>
</nav>
```

**2) Estilo base con Flexbox** Configuramos la barra como un contenedor flex en dirección horizontal:

```
.nav {
  display: flex;      /* coloca los enlaces en fila */
  gap: 12px;         /* espacio uniforme entre ellos */
}
```

**3) Aplicar la media query** Cuando el ancho de la ventana es inferior a 640px, cambiamos la dirección de los elementos:

```
@media (max-width: 640px) {  
  .nav {  
    flex-direction: column; /* cambia a columna */  
  }  
}
```

### Código completo

```
1 <nav class="nav">  
2   <a>Inicio</a><a>Blog</a><a>Contacto</a>  
3 </nav>  
4  
5 <style>  
6 .nav {  
7   display: flex;  
8   gap: 12px;  
9 }  
10 @media (max-width: 640px) {  
11   .nav {  
12     flex-direction: column;  
13   }  
14 }  
15 </style>
```

### Conclusión

- Flexbox permite una disposición flexible de los elementos.
- La media query garantiza que la interfaz se adapte a distintos tamaños de pantalla.
- Con esta combinación logramos una barra de navegación responsiva mínima: horizontal en escritorio, vertical en móvil.

### 8.8. Ejercicio Transición y estado :hover/:focus-visible

Crea botones con transición suave de color y elevación (**transform**) al interactuar con ratón/teclado. **Solución.**

```
1 .button{  
2   background:#2D6CDF; color:#fff; padding:.75rem 1rem;  
3   border-radius:.5rem; display:inline-block;  
4   transition: background .25s ease, transform .2s ease, box-shadow .2s  
5     ease;  
6 }  
7 .button:hover,  
8 .button:focus-visible{  
9   background:#1f4fa3; transform: translateY(-2px);  
10  box-shadow:0 6px 18px rgba(0,0,0,.15);  
11 }
```

Transiciones mejoran UX; **:focus-visible** mantiene accesibilidad. (Amplía temario con pseudo-clases y transiciones).

### 8.9. Ejercicio Flexbox: alineación de tarjetas

Tres tarjetas en fila, con misma altura, botón pegado abajo, y separación uniforme. (1.4)

## Teoría esencial

- **Transiciones:** permiten que los cambios de estilo sucedan de forma suave a lo largo del tiempo. Se definen con la propiedad `transition`, indicando qué propiedades se animan, su duración y la función de tiempo (p.ej., `ease`, `linear`, `ease-in-out`).
- **Pseudoclases `:hover` y `:focus-visible`:**
  - `:hover` se activa al pasar el ratón sobre el elemento.
  - `:focus-visible` se activa cuando el elemento tiene foco mediante teclado u otros métodos de accesibilidad, mostrando estilos sin necesidad de ratón.
- **Transformaciones:** con `transform` podemos desplazar, rotar o escalar elementos. En este caso se aplica un ligero `translateY` para simular elevación del botón.
- **Sombra proyectada:** `box-shadow` refuerza la sensación de elevación en el estado activo.

## Paso a paso

**1) Estilo base del botón** Se define el color de fondo, color de texto, relleno, bordes redondeados y se especifica la transición:

```
.button {  
  background: #2D6CDF;  
  color: #fff;  
  padding: .75rem 1rem;  
  border-radius: .5rem;  
  display: inline-block;  
  transition: background .25s ease,  
             transform .2s ease,  
             box-shadow .2s ease;  
}
```

**2) Estilo en interacción** Al pasar el ratón o enfocar con teclado, el botón cambia de color, se eleva y proyecta sombra:

```
.button:hover,  
.button:focus-visible {  
  background: #1f4fa3;  
  transform: translateY(-2px);  
  box-shadow: 0 6px 18px rgba(0,0,0,.15);  
}
```

## Código completo

```
1 .button {  
2   background: #2D6CDF;  
3   color: #fff;  
4   padding: .75rem 1rem;  
5   border-radius: .5rem;  
6   display: inline-block;  
7   transition: background .25s ease,  
8             transform .2s ease,  
9             box-shadow .2s ease;  
10 }  
11 .button:hover,  
12 .button:focus-visible {
```

```

13 | background: #1f4fa3;
14 | transform: translateY(-2px);
15 | box-shadow: 0 6px 18px rgba(0,0,0,.15);
16 | }

```

## Conclusión

- Las transiciones suavizan la experiencia de usuario al hacer los cambios visuales más agradables.
- El uso combinado de `:hover` y `:focus-visible` garantiza tanto interactividad con ratón como accesibilidad con teclado.
- La combinación de color, desplazamiento y sombra refuerza la metáfora de botón interactivo y mejora la usabilidad.

## 8.10. Ejercicio Grid: layout holy-grail simple

Construye un layout con cabecera, barra lateral, contenido y pie; que la barra se sitúe a la izquierda en escritorio y abajo en móvil.

### Teoría esencial

- **CSS Grid:** sistema de maquetación bidimensional que permite organizar contenido en filas y columnas, definiendo áreas semánticas y controlando la disposición de los elementos sin modificar el HTML.
- **Layout holy-grail:** patrón clásico de maquetación con cuatro zonas: cabecera, barra lateral, contenido principal y pie. En escritorio la barra suele situarse a la izquierda; en móvil, bajo el contenido.
- **Media queries:** posibilitan adaptar la distribución del grid según el ancho de pantalla, manteniendo un diseño responsivo.

### Paso a paso

**1) Definir el contenedor principal** Creamos un contenedor con `display:grid`, separaciones uniformes y una disposición inicial de escritorio:

```

.layout {
  display: grid;
  gap: 12px;
  grid-template-areas:
    "header header"
    "aside main"
    "footer footer";
  grid-template-columns: 240px 1fr;
}

```

**2) Asignar cada elemento a su área** Cada bloque se vincula a un área semántica del grid:

```

.header { grid-area: header; background: #eee; }
.aside { grid-area: aside; background: #f2f2f2; }
.main { grid-area: main; background: #fff; }
.footer { grid-area: footer; background: #eee; }

```

**3) Adaptar a pantallas pequeñas** Bajo 700px de ancho, redefinimos las áreas para que la barra lateral quede debajo del contenido:

```
@media (max-width: 700px) {  
  .layout {  
    grid-template-areas:  
      "header"  
      "main"  
      "aside"  
      "footer";  
    grid-template-columns: 1fr;  
  }  
}
```

### Código completo

```
1 .layout{  
2   display:grid; gap:12px;  
3   grid-template-areas:  
4     "header header"  
5     "aside main"  
6     "footer footer";  
7   grid-template-columns: 240px 1fr;  
8 }  
9 .header{ grid-area: header; background:#eee; }  
10 .aside { grid-area: aside; background:#f2f2f2; }  
11 .main { grid-area: main; background:#fff; }  
12 .footer{ grid-area: footer; background:#eee; }  
13  
14 @media (max-width:700px){  
15   .layout{  
16     grid-template-areas:  
17       "header"  
18       "main"  
19       "aside"  
20       "footer";  
21     grid-template-columns: 1fr;  
22   }  
23 }
```

### Conclusión

- CSS Grid facilita la creación de layouts semánticos y limpios, evitando modificar el orden del HTML.
- El patrón holy-grail se implementa de forma concisa con `grid-template-areas`.
- La combinación con media queries permite un diseño responsivo adaptable a escritorio y móvil.

## 9. Ejercicios propuestos

### 9.1. Selector `:nth-child` complejo

**Enunciado** Diseña una tabla donde las filas pares tengan fondo gris claro, las filas múltiplo de 3 tengan fondo azul claro y las celdas en columnas pares estén en negrita.

**Teoría** `:nth-child(an+b)` Permite seleccionar elementos según progresiones aritméticas:  $2n$  (pares),  $3n$  (múltiplos de 3),  $2n+1$  (impares), etc.

**Ámbito de aplicación** En tablas, se suele aplicar sobre `tr` para filas y sobre `td` para columnas.

**Conflictos y cascada** Si dos reglas afectan a la misma fila (p. ej., la fila 6 es par y múltiplo de 3), prevalece la regla con mayor especificidad o, a igualdad, la declarada más abajo en el CSS.

**Paso a paso** 1 Estilizar la tabla base (`border-collapse`, márgenes, etc.).

2 Colorear las filas pares con `tr:nth-child(2n)`.

3 Colorear los múltiplos de 3 con `tr:nth-child(3n)` y declarar esta regla después para que tenga prioridad en las filas que coinciden.

4 Poner en negrita las celdas de columnas pares con `td:nth-child(2n)`.

5 Verificar filas de solapamiento (6, 12, ...) para confirmar que el azul claro prevalece sobre el gris claro.

**Notas** **Orden de reglas** Declarar la regla de  $3n$  después de  $2n$  hace que el azul claro gane en las filas 6, 12, ...

**Especificidad** Si se necesita forzar prioridad, puede aumentarse la especificidad (p. ej., `table tbody tr:nth-child(3n)`), aunque suele bastar con el orden.

**Scope** Usar `tbody` evita que `thead` o `tfoot` entren en el conteo de `:nth-child`.

```
1  table {
2    border-collapse: collapse;
3    width: 100%;
4  }
5  th, td {
6    padding: 0.5rem;
7    border: 1px solid #ddd;
8  }
9
10 /* Filas pares: gris claro */
11 tbody tr:nth-child(2n) {
12   background: #f5f5f5;
13 }
14
15 /* Múltiplos de 3: azul claro (declarado después para prevalecer) */
16 tbody tr:nth-child(3n) {
17   background: #e8f0ff;
18 }
19
20 /* Columnas pares en negrita */
21 tbody td:nth-child(2n),
22 thead th:nth-child(2n) { /* opcional: también en cabecera */
23   font-weight: 700;
24 }
```

## 9.2. Animaciones con @keyframes

**Enunciado** Crea una animación que mueva un cuadrado de izquierda a derecha de manera infinita, acelerando al inicio y frenando al final.

**Teoría Definición** Las animaciones en CSS se crean con `@keyframes`, donde se describen los estados iniciales, intermedios y finales de las propiedades.

**Propiedades transformables** Para un movimiento fluido se recomienda usar `transform: translateX()` en lugar de cambiar márgenes o posiciones absolutas, ya que los navegadores optimizan estas transformaciones.

**Función de tiempo** La velocidad de la animación se controla con `animation-timing-function`. La opción `ease-in-out` hace que la animación acelere al inicio y desacelere al final.

**Repetición** La propiedad `animation-iteration-count: infinite` garantiza que el ciclo se repita indefinidamente.

**Paso a paso 1** Crear un elemento cuadrado con dimensiones y color definidos.

- 2 Declarar la animación con `@keyframes`, indicando el punto de inicio (izquierda) y el final (derecha).
- 3 Asignar la animación al cuadrado con propiedades como `animation-name`, `animation-duration`, `animation-timing-function` y `animation-iteration-count`.
- 4 Verificar en el navegador que el movimiento sea continuo y con la aceleración y desaceleración esperadas.

```
1  div.cuadro {
2    width: 50px; height: 50px;
3    background: red;
4    animation: mover 3s ease-in-out infinite;
5  }
6
7  @keyframes mover {
8    from { transform: translateX(0); }
9    to   { transform: translateX(300px); }
10 }
```

## 9.3. Pseudo-elementos avanzados

**Enunciado** Diseña un botón que al pasar el ratón muestre una sombra interna animada usando `::after`.

**Teoría Pseudo-elementos** `::before` y `::after` permiten generar contenido adicional en CSS sin modificar el HTML.

**Posicionamiento** Para que funcionen correctamente, el elemento principal (el botón) debe tener `position: relative`, de modo que el pseudo-elemento pueda colocarse con `position: absolute`.

**Sombra interna** Se puede simular mediante `box-shadow: inset` o manipulando `opacity`.

**Transiciones** Al añadir `transition` sobre la propiedad elegida (opacidad, sombra, etc.), el efecto aparece de forma suave al pasar el ratón con `:hover`.

**Paso a paso 1** Crear un botón con estilos básicos y `position: relative`.

- 2 Definir el pseudo-elemento `::after`, ajustando ancho, alto y posición absoluta para que cubra el botón.
- 3 Aplicar `box-shadow: inset` o `opacity: 0` como estado inicial.

- 4 Añadir una **transition** sobre la propiedad que simula la sombra.
- 5 En el estado **:hover**, modificar **box-shadow** o **opacity** para que aparezca la sombra interna de forma animada.

```
1  button {
2    position: relative;
3    padding: 10px 20px;
4    background: #2D6CDF;
5    color: white;
6    border: none;
7    cursor: pointer;
8    overflow: hidden;
9  }
10
11 button::after {
12   content: "";
13   position: absolute;
14   top: 0; left: 0;
15   width: 100%; height: 100%;
16   box-shadow: inset 0 0 0 0 rgba(0,0,0,0.5);
17   transition: box-shadow 0.3s ease;
18 }
19
20 button:hover::after {
21   box-shadow: inset 0 0 10px 5px rgba(0,0,0,0.3);
22 }
```

## 9.4. Variables CSS y herencia

**Enunciado** Crea un esquema de colores donde un contenedor padre defina **--primary** y sus hijos lo usen para fondo o borde. Cambia el valor en un sub-contenedor y observa el efecto.

**Teoría Variables CSS** Se definen con **-nombre** y se usan con **var(-nombre)**.

**Herencia** Estas variables se comportan como valores heredables: los elementos descendientes pueden acceder al valor definido en un ancestro.

**Sobreescritura** Si un contenedor redefine la variable, todos sus hijos adoptan el nuevo valor sin necesidad de cambiar sus estilos internos.

**Escalabilidad** Esta técnica permite construir sistemas de diseño temáticos fácilmente mantenibles y flexibles.

**Paso a paso 1** Definir en el contenedor padre la variable **--primary** con un color.

**2** Usar dicha variable en los hijos para propiedades como **background** o **border**.

**3** Crear un sub-contenedor que redefina **--primary** con otro valor.

**4** Observar cómo todos los elementos dentro del sub-contenedor adoptan el nuevo color automáticamente.

```
1  .padre {
2    --primary: #2D6CDF;
3    padding: 20px;
4  }
5
6  .hijo {
7    background: var(--primary);
8    border: 2px solid var(--primary);
9  }
```

```

9      color: white;
10     padding: 10px;
11     margin: 5px 0;
12 }
13
14 .subcontenedor {
15     --primary: #F05454;
16     padding: 10px;
17 }

```

## 9.5. Grid con áreas solapadas (1.5)

**Enunciado** Diseña un grid 2x2 donde un elemento abarque las cuatro celdas como fondo (con opacidad), y los otros tres aparezcan encima en distintas posiciones.

**Teoría Grid y áreas** Con CSS Grid podemos hacer que un mismo elemento ocupe varias celdas mediante `grid-area`, `grid-column` o `grid-row`.

**Superposición** Los elementos del grid no solo se colocan en filas y columnas; también pueden solaparse en las mismas áreas.

**Capas** El control de qué elemento queda arriba o debajo se logra con `z-index`.

**Opacidad** Usar `opacity` en el elemento que actúa de fondo permite visualizar los contenidos superpuestos.

**Paso a paso 1** Crear un contenedor con `display: grid` y definir una cuadrícula 2x2 mediante `grid-template-columns` y `grid-template-rows`.

**2** Añadir un elemento que abarque todo el grid usando `grid-column: 1 / -1` y `grid-row: 1 / -1`.

**3** Asignar opacidad a este elemento para que funcione como fondo.

**4** Colocar otros tres elementos en distintas posiciones del grid, cada uno en su celda.

**5** Usar `z-index` mayor en los elementos superiores para garantizar que aparezcan sobre el fondo.

```

1  .grid {
2      display: grid;
3      grid-template-columns: 1fr 1fr;
4      grid-template-rows: 1fr 1fr;
5      gap: 10px;
6      width: 400px; height: 400px;
7  }
8
9  .fondo {
10     grid-column: 1 / -1;
11     grid-row: 1 / -1;
12     background: blue;
13     opacity: 0.2;
14     z-index: 0;
15 }
16
17 .item1 { background: red; z-index: 1; }
18 .item2 { background: green; z-index: 1; }
19 .item3 { background: orange; z-index: 1; }

```

## 9.6. Flexbox con order y grow

**Enunciado** Diseña un layout con 5 cajas que: la tercera ocupe el doble de espacio que las demás, y la última se muestre primera en pantallas pequeñas (`max-width:600px`).

**Teoría Flex-grow** Controla cómo se reparte el espacio sobrante entre los elementos flex. Un valor mayor implica que el elemento ocupará más proporción de espacio.

**Order** Permite cambiar el orden visual de los elementos sin modificar el HTML. Por defecto, todos tienen `order:0`. Un número menor se mostrará antes.

**Responsividad** Combinando Flexbox (1.4) con `@media queries`, es posible adaptar el orden o el crecimiento de los elementos según el ancho de la pantalla.

**Paso a paso 1** Crear un contenedor con `display: flex`.

**2** Definir 5 elementos hijos (`.caja`).

**3** Dar a todos `flex-grow:1` para ocupar el mismo espacio.

**4** Asignar a la tercera caja `flex-grow:2` para que ocupe el doble que las demás.

**5** Usar una media query con `max-width:600px` para cambiar el `order` de la última caja a `-1`, de modo que aparezca primero.

```
1  .contenedor {
2      display: flex;
3      gap: 10px;
4  }
5
6  .caja {
7      flex-grow: 1;
8      background: lightgray;
9      padding: 20px;
10     text-align: center;
11 }
12
13 .caja:nth-child(3) {
14     flex-grow: 2;
15     background: lightblue;
16 }
17
18 @media (max-width:600px) {
19     .caja:nth-child(5) {
20         order: -1;
21         background: lightgreen;
22     }
23 }
```

## 9.7. Clipping y máscaras

**Enunciado** Crea una tarjeta con imagen circular usando `clip-path: circle()`, y añade una transición que cambie a forma hexagonal al pasar el ratón.

**Teoría Clip-path** Permite recortar la región visible de un elemento en formas básicas (`circle()`, `ellipse()`, `inset()`) o personalizadas con `polygon()`.

**Animación** Al aplicar `transition` sobre `clip-path`, los navegadores pueden interpolar entre formas compatibles; en la práctica, suele funcionar entre `circle()` y `polygon()` simples para este tipo de efectos.

**Hexágono** Un `polygon()` de seis puntos simula un hexágono. Ajustando los vértices se controla el radio y la redondez aparente.

**Compatibilidad** En algunos entornos puede requerirse el prefijo `-webkit-clip-path`. Añadir `will-change: clip-path` puede mejorar la suavidad.

**Paso a paso 1** Crear una tarjeta contenedor con tamaño fijo y `overflow: hidden`.

**2** Insertar una imagen que cubra toda la tarjeta con `object-fit: cover`.

**3** Aplicar `clip-path: circle()` a la imagen para que se vea circular por defecto.

**4** Definir `transition: clip-path 0.4s ease-in-out`.

**5** En `:hover` del contenedor, cambiar a `clip-path: polygon()` con seis puntos (hexágono).

```
1  .card {
2    width: 280px; height: 280px;
3    border-radius: 16px;
4    overflow: hidden;
5    position: relative;
6    display: grid; place-items: center;
7    background: #f5f7fb;
8  }
9
10 .card img {
11   width: 100%; height: 100%;
12   object-fit: cover;
13   clip-path: circle(45% at 50% 50%);
14   -webkit-clip-path: circle(45% at 50% 50%);
15   transition: clip-path 0.4s ease-in-out,
16             -webkit-clip-path 0.4s ease-in-out;
17   will-change: clip-path;
18 }
19
20 /* Hexágono regular aproximado centrado */
21 .card:hover img {
22   clip-path: polygon(
23     50% 5%,
24     93% 27%,
25     93% 73%,
26     50% 95%,
27     7% 73%,
28     7% 27%
29   );
30   -webkit-clip-path: polygon(
31     50% 5%,
32     93% 27%,
33     93% 73%,
34     50% 95%,
35     7% 73%,
36     7% 27%
37   );
38 }
```

## 9.8. Tipografía responsiva avanzada

**Enunciado** Crea un título que se mantenga entre `2rem` y `4rem`, pero escale proporcionalmente con el ancho de la pantalla usando `clamp()`.

**Teoría** `clamp(min, preferido, max)` Limita un valor fluido entre un mínimo y un máximo. En tipografía, el término preferido suele ser un tamaño dependiente del viewport (p. ej., `vw`) para crecer/disminuir de forma continua.

**Límites seguros** `min` evita que el texto quede demasiado pequeño; `max` evita tamaños excesivos. Así se obtiene legibilidad estable en todo rango de dispositivos.

**Fluidez** Un preferido como `5vw` hace que el tamaño escale con el ancho; `clamp` acota ese crecimiento entre `2rem` y `4rem`.

**Paso a paso 1** Definir el selector del título (p. ej., `h1` o una clase).

**2** Establecer `font-size` con `clamp(2rem, X, 4rem)`, donde `X` sea un valor fluido (p. ej., `5vw`).

**3** Ajustar `X` hasta lograr una progresión agradable entre los anchos de pantalla típicos (móvil → escritorio).

**4** Añadir `line-height` y márgenes coherentes para mantener ritmo vertical.

**Variantes útiles Preferido mixto** Usar `calc()` para combinar base fija y parte fluida: `clamp(2rem, calc(1rem + 4vw), 4rem)`.

**Microajustes** Si el crecimiento es muy rápido/lento, reduce/aumenta el porcentaje de `vw`.

```
1  /* Título que fluye entre 2rem y 4rem */
2  h1.titulo-responsivo {
3      font-size: clamp(2rem, 5vw, 4rem);
4      line-height: 1.1;
5      margin: 0.5em 0;
6  }
```

## 9.9. Animación con variables CSS

**Enunciado** Define un gradiente con ángulo en una variable `-angle`, y anima ese ángulo para crear un fondo con gradiente rotatorio infinito.

**Teoría Variables CSS** Las custom properties pueden almacenar valores numéricos, como un ángulo en grados.

**Gradientes dinámicos** Al usar `var(-angle)` dentro de `linear-gradient()`, el gradiente adopta el ángulo definido en la variable.

**Interpolación** Los navegadores interpolan valores numéricos en animaciones de variables CSS, por lo que `-angle` puede pasar de `0deg` a `360deg` de manera continua.

**Ventaja** Se logra un gradiente animado sin imágenes ni JavaScript, solo con CSS moderno.

**Paso a paso 1** Definir en el selector raíz la variable `-angle` con un valor inicial.

**2** Crear un fondo con `linear-gradient(var(-angle), color1, color2)`.

**3** Definir un `@keyframes` que anime la variable `-angle` de `0deg` a `360deg`.

**4** Asignar la animación al elemento, configurando duración, repetición infinita y función de tiempo lineal.

**Resumen** Animar variables CSS permite transformar gradientes de forma fluida. Con `@keyframes` aplicado sobre `-angle`, el gradiente rota indefinidamente, creando un efecto visual dinámico y moderno sin necesidad de recursos externos.

```
1  :root {
2      --angle: 0deg;
3  }
4
5  body {
6      height: 100vh;
7      display: grid; place-items: center;
```

```

8     background: linear-gradient(var(--angle), #2D6CDF, #F05454);
9     animation: girar 8s linear infinite;
10 }
11
12 @keyframes girar {
13     from { --angle: 0deg; }
14     to   { --angle: 360deg; }
15 }

```

## 9.10. Accesibilidad y media queries avanzadas

**Enunciado** Diseña un esquema de colores que se adapte a `prefers-color-scheme: dark`, y otro que reduzca las animaciones si el usuario tiene `prefers-reduced-motion`.

**Teoría Preferencias del usuario** Las media queries `prefers-color-scheme` y `prefers-reduced-motion` consultan ajustes del sistema/navegador para respetar la accesibilidad.

**Tema `prefers-color-scheme`** permite cambiar paletas entre claro/oscuro sin depender de clases ni JS.

**Movimiento `prefers-reduced-motion`** sugiere minimizar, simplificar o desactivar animaciones/transiciones potencialmente molestas.

**Estrategia** Define valores por defecto (tema claro y animaciones normales) y sobrescribe dentro de cada media query para oscurecer paleta o reducir movimiento.

**Paso a paso 1** Declarar variables de color en `:root` para el tema claro por defecto.

**2** Usar `@media (prefers-color-scheme: dark)` para redefinir las mismas variables con una paleta oscura.

**3** Referenciar las variables en tus componentes (`background`, `color`, `border`, etc.).

**4** Añadir animaciones/transiciones al contenido que lo requiera.

**5** En `@media (prefers-reduced-motion: reduce)`, desactivar o simplificar animaciones y transiciones (p.ej., `animation: none`, `transition: none`).

```

1  /* 1) Valores por defecto (claro) */
2  :root{
3      --bg: #ffffff;
4      --fg: #222222;
5      --prim: #2D6CDF;
6  }
7
8  /* 2) Tema oscuro respetando el S0 */
9  @media (prefers-color-scheme: dark) {
10     :root{
11         --bg: #0f1115;
12         --fg: #e8eaed;
13         --prim: #8ab4f8;
14     }
15 }
16
17 /* 3) Uso de tokens en la UI */
18 body { background: var(--bg); color: var(--fg); }
19 a { color: var(--prim); text-decoration: underline; }
20
21 /* 4) Un ejemplo con animación/transición */
22 .panel {
23     transform: translateY(0);
24     transition: transform 300ms ease, opacity 300ms ease;

```

```
25 }
26 .panel--entrando {
27   transform: translateY(-8px);
28   opacity: 0.9;
29 }
30
31 /* 5) Reducir movimiento si el usuario lo prefiere */
32 @media (prefers-reduced-motion: reduce) {
33   * { scroll-behavior: auto; }
34   .panel {
35     transition: none;
36   }
37   .panel--entrando {
38     transform: none;
39     opacity: 1; /* alternativa sin animación */
40   }
41   /* Anula cualquier animación declarada en otro lugar */
42   .animable { animation: none !important; }
43 }
```