CSS

Cascading Style Sheets

- White Space und Text-Wrapping in CSS – eine ausführliche Anleitung
- SASS Mixin für typografische Kontrolle in Bricks Builder
- OKLCH
  - HSL-L ist nicht „wahrgenommene Helligkeit“
  - OKLCH - SCSS/SASS - Dark/Light Mode
  - HSL lightness is not true perceived brightness
  - Ways to define colors in CSS — from HEX to OKLCH
- Dark Mode
  - Dark Mode: Gängige Methoden

White Space und Text-Wrapping in CSS – eine ausführliche Anleitung

Wenn es um Textdarstellung in CSS geht, sind zwei Themen besonders wichtig:

1. **Wie Leerzeichen, Zeilenumbrüche und Tabulatoren behandelt werden**
2. **Wie und wo Text umbrochen werden darf**

Genau dafür gibt es in CSS mehrere Eigenschaften rund um **White Space** und **Text Wrap**. Manche davon sind altbekannt, andere stammen aus moderneren CSS-Spezifikationen. Zusammen steuern sie, ob Text in einer Zeile bleibt, umbricht, Leerzeichen zusammenfasst, lange Wörter trennt oder über den Container hinausragt.

Überblick: Was gehört thematisch dazu?

Die wichtigsten CSS-Eigenschaften in diesem Bereich sind:

1. `white-space`
2. `overflow-wrap`
3. `word-break`
4. `line-break`
5. `hyphens`
6. `text-wrap`
7. `tab-size`

Außerdem gibt es verwandte Themen, die oft damit verwechselt oder gemeinsam eingesetzt werden:

1. `writing-mode`
2. `direction`

3. `text-overflow`
4. `overflow`
5. `display` und verfügbare Breite des Elements

Das Grundproblem: Warum braucht man diese Eigenschaften?

Standardmäßig behandelt der Browser normalen Fließtext ungefähr so:

- Mehrere Leerzeichen hintereinander werden meist zu **einem** Leerzeichen zusammengefasst.
- Zeilenumbrüche im HTML-Quelltext werden meist nicht als sichtbare neue Zeilen dargestellt.
- Text darf an geeigneten Stellen automatisch umbrechen.
- Sehr lange Wörter oder URLs können Probleme machen, wenn kein sinnvoller Umbruchpunkt existiert.

Beispiel:

```
<p>
  Das    ist      Text
  mit mehreren Leerzeichen
  und einem Zeilenumbruch im HTML.
</p>
```

Ohne besondere CSS-Regeln wird das im Browser ungefähr wie ein normaler Satz dargestellt – also nicht mit exakt denselben Leerzeichen und Zeilenumbrüchen wie im Quelltext.

`white-space` – die zentrale Eigenschaft

Die wichtigste Eigenschaft für White Space ist `white-space`. Sie steuert vor allem:

- ob Leerzeichen zusammengefasst werden
- ob Zeilenumbrüche aus dem Quelltext erhalten bleiben
- ob automatischer Zeilenumbruch erlaubt ist

Syntax

```
.element {
  white-space: normal;
}
```

Die wichtigsten Werte

`white-space: normal`

Das ist das Standardverhalten.

- Mehrere Leerzeichen werden zusammengefasst.
- Zeilenumbrüche im Quelltext werden ignoriert bzw. wie normale Leerzeichen behandelt.
- Automatischer Zeilenumbruch ist erlaubt.

```
p {
  white-space: normal;
}
```

Typischer Einsatz: normaler Fließtext.

`white-space: nowrap`

- Leerzeichen werden weiterhin zusammengefasst.
- Zeilenumbrüche aus dem Quelltext werden nicht als echte neue Zeilen behandelt.
- **Automatischer Zeilenumbruch wird verhindert.**

```
.badge {
  white-space: nowrap;
}
```

Effekt: Der gesamte Text bleibt in einer Zeile, sofern nicht explizit ein `<br>` oder ähnliches vorhanden ist.

Typische Einsätze:

- Buttons
- Labels
- Tabellenzellen
- Navigationseinträge
- „Nicht umbrechen“-Bereiche wie Preisangaben oder Icons mit Text

Achtung: In schmalen Containern kann das zu horizontalem Overflow führen.

`white-space: pre`

Dieses Verhalten ähnelt dem HTML-Element `<pre>`.

- Leerzeichen bleiben erhalten.
- Zeilenumbrüche bleiben erhalten.
- Automatischer Umbruch findet **nicht** statt.

```
pre,
.code-like {
  white-space: pre;
}
```

Typischer Einsatz:

- Codeblöcke
- ASCII-Layouts
- Inhalte, bei denen exakte Einrückung relevant ist

`white-space: pre-wrap`

- Leerzeichen bleiben erhalten.
- Zeilenumbrüche bleiben erhalten.
- Automatischer Zeilenumbruch ist **zusätzlich** erlaubt.

```
.message {
  white-space: pre-wrap;
}
```

Das ist sehr praktisch für Inhalte wie:

- Benutzereingaben
- Chat-Nachrichten
- Kommentare
- Texte aus Textareas

Denn damit bleiben manuelle Zeilenumbrüche erhalten, aber lange Zeilen können dennoch umbrechen.

`white-space: pre-line`

- Mehrere Leerzeichen werden zusammengefasst.
- Zeilenumbrüche bleiben erhalten.
- Automatischer Zeilenumbruch ist erlaubt.

```
.poem {
  white-space: pre-line;
}
```

Unterschied zu `pre-wrap`:

- `pre-wrap` bewahrt auch mehrere Leerzeichen
- `pre-line` bewahrt nur Zeilenumbrüche, aber nicht die exakte Anzahl von Leerzeichen

---

`white-space: break-spaces`

Ein moderner Wert mit sehr speziellem Verhalten.

- Leerzeichen bleiben erhalten.
- Zeilenumbrüche bleiben erhalten.
- Umbruch kann auch an erhaltenen Leerzeichen stattfinden.
- Nachfolgende Leerzeichen am Zeilenende bleiben relevant.

```
.output {
  white-space: break-spaces;
}
```

Dieser Wert ist nützlich, wenn wirklich die **sichtbare Struktur von Leerzeichen** wichtig ist.

Vergleichstabelle zu `white-space`

Wert	Leerzeichen erhalten?	Zeilenumbrüche erhalten?	automatischer Umbruch?
`normal`	Nein	Nein	Ja
`nowrap`	Nein	Nein	Nein
`pre`	Ja	Ja	Nein
`pre-wrap`	Ja	Ja	Ja
`pre-line`	Nein	Ja	Ja
`break-spaces`	Ja	Ja	Ja

Moderne Aufteilung: White-Space-Untereigenschaften

In neueren CSS-Spezifikationen wird `white-space` konzeptionell in feinere Teilaspekte aufgeteilt. Dazu gehören unter anderem:

1. `white-space-collapse`
2. `text-wrap-mode`
3. `white-space-trim`

Diese sind konzeptionell wichtig, aber **noch nicht überall gleich gut etabliert** wie `white-space`. In der Praxis verwendet man deshalb meist weiterhin `white-space`.

`white-space-collapse`

Diese Eigenschaft steuert, wie Leerraum zusammengefasst oder erhalten wird.

Mögliche Werte sind je nach Spezifikation unter anderem:

- `collapse`
- `preserve`
- `preserve-breaks`

- `preserve-spaces`
- `break-spaces`

Beispielidee:

```
.element {
  white-space-collapse: preserve;
}
```

Praxis-Hinweis: Diese Eigenschaft ist interessant für moderne CSS-Modelle, aber für produktive, breit kompatible Websites ist `white-space` oft die sicherere Wahl.

`text-wrap-mode`

Sie beschreibt grundsätzlich, ob Zeilenumbruch erlaubt ist.

Typische Werte:

- `wrap`
- `nowrap`

Beispiel:

```
.element {
  text-wrap-mode: nowrap;
}
```

Auch hier gilt: In der Praxis ist meist `white-space: nowrap;` der bekanntere und breiter eingesetzte Weg.

`white-space-trim`

Diese Eigenschaft soll beeinflussen, ob bestimmte Whitespaces an Anfang oder Ende entfernt werden.

Beispielhaft konzeptionell:

```
.element {
  white-space-trim: discard-before;
```

```
}
```

Auch das ist eher ein fortgeschrittenes bzw. modernes Thema mit eingeschränkter Relevanz im Alltag.

`overflow-wrap` – was passiert mit langen Wörtern?

`overflow-wrap` bestimmt, ob der Browser **lange Wörter oder Zeichenketten umbrechen darf**, wenn sie sonst den Container sprengen würden.

Früher war dafür oft `word-wrap` im Einsatz. Das ist heute im Grunde ein Alias für `overflow-wrap`.

Syntax

```
.element {
  overflow-wrap: normal;
}
```

Werte

`overflow-wrap: normal`

Der Browser bricht nur an normalen Umbruchstellen um.

```
.element {
  overflow-wrap: normal;
}
```

Lange URLs oder zusammengesetzte Wörter können dann überlaufen.

`overflow-wrap: break-word`

Falls nötig, darf ein langes Wort umgebrochen werden, um Overflow zu verhindern.

```
.article {
  overflow-wrap: break-word;
}
```

Typischer Einsatz:

- CMS-Inhalte
- Foren
- Kommentare
- UGC („user generated content“)
- lange URLs

`overflow-wrap: anywhere`

Noch aggressiver: Der Browser darf an praktisch jeder Stelle umbrechen, wenn nötig.

```
.article {
  overflow-wrap: anywhere;
}
```

Wann sinnvoll?

- bei extrem langen Tokens
- bei technischen IDs
- bei URLs ohne sinnvolle Trennpunkte
- in sehr schmalen Layouts

Unterschied zu `break-word`:
`anywhere` erlaubt Umbrüche sehr frei und berücksichtigt diese Möglichkeiten auch stärker bei der Zeilenberechnung.

`word-wrap` – historischer Alias

```
.element {
  word-wrap: break-word;
```

```
}
```

Das funktioniert oft noch, aber modern und sauber ist:

```
.element {
  overflow-wrap: break-word;
}
```

`word-break` – wie aggressiv darf in Wörtern gebrochen werden?

`word-break` beeinflusst den Umbruch **innerhalb von Wörtern** stärker als `overflow-wrap`.

Syntax

```
.element {
  word-break: normal;
}
```

Werte

`word-break: normal`

Normales Umbruchverhalten nach Sprach- und Schriftsystemregeln.

```
.element {
  word-break: normal;
}
```

`word-break: break-all`

Wörter dürfen praktisch an beliebigen Stellen getrennt werden.

```
.element {
  word-break: break-all;
}
```

Effekt: Auch normale Wörter können mitten im Wort umbrechen.

Das löst Overflow-Probleme zuverlässig, sieht aber oft unschön aus.

Einsatz nur mit Vorsicht, z. B. bei:

- sehr schmalen technischen Layouts
- Tabellen mit langen Schlüsseln
- maschinenlesbaren Zeichenfolgen

`word-break: keep-all`

Verhindert Wortumbrüche innerhalb von Wörtern, besonders relevant für ostasiatische Schriftsysteme.

```
.element {
  word-break: keep-all;
}
```

Für deutschsprachige Seiten ist dieser Wert seltener relevant, kann aber in internationalen Projekten wichtig sein.

Unterschied zwischen `overflow-wrap` und `word-break`

Das ist einer der häufigsten Stolperpunkte.

`overflow-wrap`

- greift vor allem dann, wenn ein Wort **sonst überlaufen würde**
- ist eher eine „Notfalllösung“

`word-break`

- beeinflusst allgemeiner, **wie innerhalb von Wörtern getrennt werden darf**
- ist meist aggressiver

Faustregel

1. Erst `overflow-wrap` prüfen
2. Nur wenn das nicht reicht, `word-break` einsetzen
3. `break-all` nur bewusst und sparsam verwenden

`hyphens` – automatische Silbentrennung

Mit `hyphens` kann der Browser Wörter an geeigneten Stellen trennen – oft mit Bindestrich.

Syntax

```
.element {
  hyphens: auto;
}
```

Werte

`hyphens: none`

Keine Silbentrennung.

```
.element {
  hyphens: none;
}
```

`hyphens: manual`

Nur manuell vorgegebene Trennstellen werden verwendet.

```
.element {
  hyphens: manual;
}
```

Manuelle Trennstellen können z. B. im HTML gesetzt werden.

`hyphens: auto`

Automatische Silbentrennung nach Sprachregeln, sofern der Browser und die Spracheinstellungen das unterstützen.

```
p {
  hyphens: auto;
}
```

Wichtig ist dabei oft ein korrekt gesetztes `lang`-Attribut:

```
<html lang="de">
```

oder

```
<p lang="de">Donaudampfschifffahrtsgesellschaftskapitän</p>
```

Ohne passende Sprache kann automatische Silbentrennung unzuverlässig sein.

Wann ist `hyphens` nützlich?

- schmale Textspalten
- Magazine-Layouts
- lange deutsche Wörter
- bessere Blocksatzdarstellung

Beispiel

```
.article-text {
  hyphens: auto;
  overflow-wrap: normal;
}
```

Das führt oft zu schöneren Ergebnissen als brutal mit `word-break: break-all` zu arbeiten.

`line-break` – Regeln für Zeilenumbrüche, besonders in asiatischen Schriften

`line-break` steuert die Strenge der Umbruchregeln, insbesondere für chinesische, japanische und koreanische Texte.

Syntax

```
.element {
  line-break: auto;
}
```

Typische Werte

- `auto`
- `loose`
- `normal`
- `strict`
- `anywhere`

Beispiel:

```
.element {
  line-break: strict;
}
```

Für deutschsprachige Seiten ist diese Eigenschaft meist **nicht zentral**, aber in mehrsprachigen Projekten kann sie wichtig sein.

`text-wrap` – moderne Steuerung für Zeilenumbruch

`text-wrap` ist eine modernere Eigenschaft für Strategien des Textumbruchs.

Syntax

```
.element {
  text-wrap: wrap;
}
```

Mögliche Werte

Je nach aktuellem Implementierungsstand sind insbesondere diese relevant:

- `wrap`

- `nowrap`
- `balance`
- `pretty`
- `stable`

Nicht jeder Wert wird in jedem Browser gleich unterstützt.

`text-wrap: wrap`

Normales Umbruchverhalten.

```
.element {
  text-wrap: wrap;
}
```

`text-wrap: nowrap`

Kein automatischer Umbruch.

```
.element {
  text-wrap: nowrap;
}
```

Praktisch ähnlich zu `white-space: nowrap`, aber konzeptionell moderner auf den Umbruch fokussiert.

`text-wrap: balance`

Versucht, Zeilen ausgewogener zu verteilen. Besonders nützlich für Überschriften.

```
h1, h2 {
  text-wrap: balance;
}
```

Sehr sinnvoll für:

- Headlines

- Hero-Texte
- Card-Titel

Statt einer sehr langen und einer sehr kurzen Zeile versucht der Browser, ein harmonischeres Ergebnis zu erzeugen.

`text-wrap: pretty`

Zielt auf optisch angenehmere Umbrüche ab und versucht unschöne Ein-Zeilen-Wörter oder ungünstige Brüche zu vermeiden.

```
p {
  text-wrap: pretty;
}
```

Die Unterstützung kann je nach Browserstand variieren.

`text-wrap: stable`

Soll Umbruchverhalten stabil halten, z. B. bei bearbeitbaren Inhalten. Auch das ist eher ein fortgeschrittenes Thema.

`tab-size` – Breite von Tabulatoren

Wenn ein Text Tab-Zeichen enthält, kann `tab-size` deren visuelle Breite bestimmen.

Syntax

```
.element {
  tab-size: 4;
```

```
}
```

oder

```
.element {
  tab-size: 2;
}
```

Beispiel

```
pre {
  white-space: pre;
  tab-size: 4;
}
```

Das ist vor allem für Code, Logs oder vorformatierte Texte nützlich.

Wichtige verwandte Eigenschaften

`text-overflow`

Wenn Text nicht umbrochen wird und der Container zu klein ist, kann `text-overflow` festlegen, wie abgeschnittener Text dargestellt wird.

Typisch:

```
.ellipsis {
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

Das erzeugt die bekannte Darstellung mit „…“.

Wichtig: `text-overflow` funktioniert typischerweise nur in Kombination mit:

1. begrenzter Breite
2. `overflow: hidden`
3. meist `white-space: nowrap`

`overflow`

Wenn Text nicht in den Container passt, beeinflusst `overflow`, ob er sichtbar bleibt, abgeschnitten wird oder Scrollbars erscheinen.

```
.box {
  overflow: auto;
}
```

Mögliche Werte:

- `visible`
- `hidden`
- `clip`
- `scroll`
- `auto`

`display` und Breite

Textumbruch hängt nicht nur von Text-Eigenschaften ab, sondern auch davon, **ob überhaupt eine begrenzte Breite existiert**.

Beispiel:

```
.inline-label {
  display: inline;
}
```

Ein rein inline dargestelltes Element verhält sich anders als ein Block mit fester oder maximaler Breite.

Oft braucht man für sichtbaren Umbruch:

```
.card-title {
  display: block;
  max-width: 20rem;
}
```

Typische Praxisrezepte □□

1. Normaler Fließtext

```
p {
  white-space: normal;
  overflow-wrap: break-word;
  hyphens: auto;
}
```

Gut für: Artikel, Blogposts, CMS-Inhalte

Vorteil: normale Darstellung, aber robuste Behandlung langer Wörter.

---

2. Lange URLs oder unkontrollierter User-Content

```
.user-content {
  overflow-wrap: anywhere;
}
```

Gut für: Kommentare, Foren, Chat, Markdown-Content

---

3. Eine Zeile mit Auslassungspunkten

```
.one-line-ellipsis {
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

---

4. Vorformatierter Text oder Code

```
pre {
  white-space: pre;
  tab-size: 4;
  overflow: auto;
}
```

Wenn Code stattdessen umbrechen soll:

```
pre.wrap {
  white-space: pre-wrap;
  overflow-wrap: anywhere;
}
```

---

5. Benutzereingaben mit erhaltenen Zeilenumbrüchen

```
.user-message {
  white-space: pre-wrap;
  overflow-wrap: break-word;
```

```
}
```

Das ist ein sehr typisches und sinnvolles Setup.

---

6. Schöne Überschriften umbrechen

```
h1, h2, h3 {
  text-wrap: balance;
}
```

Fallback-orientiert kann man einfach zusätzlich auf normales Verhalten vertrauen, falls der Browser den Wert nicht unterstützt.

Häufige Stolperfallen

1. `nowrap` „funktioniert zu gut“

```
.badge {
  white-space: nowrap;
}
```

Dann bleibt wirklich alles in einer Zeile. Wenn der Container schmal ist, läuft der Text möglicherweise heraus.

Lösung: Nur dort einsetzen, wo es wirklich gewünscht ist.

---

2. `text-overflow: ellipsis` zeigt keine Punkte

Nur diese Regel reicht nicht:

```
.element {
  text-overflow: ellipsis;
}
```

Meist braucht man zusätzlich:

```
.element {
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

Und oft auch eine definierte Breite oder max. Breite.

---

3. Lange Wörter brechen trotzdem nicht um

Wenn ein langes Wort oder eine URL herausragt, hilft oft:

```
.element {
  overflow-wrap: break-word;
}
```

oder noch robuster:

```
.element {
  overflow-wrap: anywhere;
}
```

---

4. `hyphens: auto` bringt nichts

Dann fehlt oft eines der folgenden Dinge:

1. Das richtige `lang`-Attribut
2. Browser-Unterstützung
3. Geeigneter Fließtext-Kontext mit tatsächlichem Umbruchbedarf

---

5. In Flex- oder Grid-Layouts bricht Text nicht wie erwartet um

Das Problem liegt oft **nicht** an `white-space`, sondern an den Mindestgrößen der Items.

Ein klassischer Fall:

```
.flex-item {
  overflow-wrap: break-word;
}
```

Und trotzdem bricht der Text nicht schön um.

Dann kann zusätzlich nötig sein:

```
.flex-item {
  min-width: 0;
}
```

Oder in Grid-Layouts ebenfalls eine passende Größenlogik.

Das ist ein sehr häufiger Praxisfehler.

Unterschiede kurz und prägnant zusammengefasst

Wenn du Leerzeichen und Zeilenumbrüche aus dem Quelltext steuern willst

→ `white-space`

Wenn lange Wörter oder URLs den Container sprengen

→ `overflow-wrap`

Wenn innerhalb von Wörtern aggressiver getrennt werden soll

→ `word-break`

Wenn automatische Silbentrennung gewünscht ist

→ `hyphens`

Wenn Überschriften schöner umbrechen sollen

→ `text-wrap: balance`

Wenn Tabs in vorformatiertem Text korrekt aussehen sollen

→ `tab-size`

Empfehlenswerte Standardstrategien

Für normalen Content

```
.content {
  white-space: normal;
  overflow-wrap: break-word;
  hyphens: auto;
}
```

Für User-Generated Content

```
.user-content {
  white-space: pre-wrap;
  overflow-wrap: anywhere;
}
```

Das bewahrt Eingabe-Zeilenumbrüche und verhindert Layoutbruch durch lange Tokens.

Für Titel

```
.title {
  text-wrap: balance;
}
```

Für einzeilige UI-Elemente

```
.chip {
  white-space: nowrap;
}
```

Für abgeschnittene Ein-Zeilen-Texte

```
.truncate {
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

Beispiel: alles in einer kleinen Demo

```
<div class="demo">
  <p class="normal">
    Das ist ein normaler Fließtext mit einer
```

```
sehrlangenbeispielzeichenketteohnesinnvollenumbruch.
  </p>

  <p class="message">
    Hallo!
    Dies ist eine Nachricht
    mit manuellem Zeilenumbruch.
  </p>

  <p class="truncate">
    Dies ist ein sehr langer Titel, der in einer Zeile abgeschnitten werden soll.
  </p>

  <pre class="code">function test() {
\tconsole.log("Hallo");
}</pre>
</div>
```

```
.demo {
  max-width: 20rem;
  font-family: system-ui, sans-serif;
}

.normal {
  white-space: normal;
  overflow-wrap: break-word;
  hyphens: auto;
}

.message {
  white-space: pre-wrap;
  overflow-wrap: break-word;
}

.truncate {
  white-space: nowrap;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

```
.code {
  white-space: pre;
  tab-size: 4;
  overflow: auto;
}
```

Browser-Kompatibilität – was ist besonders sicher?

Sehr gut etabliert und in der Praxis „sicher“:

- `white-space`
- `overflow-wrap`
- `word-break`
- `hyphens` mit gewissen sprach- und browserabhängigen Unterschieden
- `tab-size`
- `text-overflow`

Moderner bzw. teils mit Vorsicht zu prüfen:

- `text-wrap` mit Werten wie `balance`, `pretty`, `stable`
- Untereigenschaften wie `white-space-collapse`, `text-wrap-mode`, `white-space-trim`

Empfehlung: Für produktive Projekte moderne Features gern nutzen, aber mit sinnvollen Fallbacks.

Eine praktische Merkhilfe □□

Du kannst dir das so merken:

1. **`white-space`**
   „Wie behandle ich Leerzeichen und Zeilenumbrüche grundsätzlich?“
2. **`overflow-wrap`**
   „Was mache ich mit langen Wörtern, wenn sie sonst überlaufen?“
3. **`word-break`**
   „Darf ich Wörter auch mitten drin zerlegen?“

4. `hyphens`
   „Kann ich Wörter sprachgerecht trennen?“
5. `text-wrap`
   „Welche Strategie soll der Browser für den Umbruch wählen?“
6. `tab-size`
   „Wie breit sind Tabs in vorformatiertem Text?“

Fazit

Die wichtigste Eigenschaft für White Space ist `white-space`. Sie legt die Basis dafür, wie Text, Leerzeichen und Zeilenumbrüche verarbeitet werden. Für robuste Layouts reicht das allein aber oft nicht aus.

In der Praxis ist diese Kombination besonders häufig sinnvoll:

```
.element {
  white-space: normal;
  overflow-wrap: break-word;
  hyphens: auto;
}
```

Für Benutzereingaben dagegen oft eher:

```
.element {
  white-space: pre-wrap;
  overflow-wrap: break-word;
}
```

Und für moderne, schön umbrechende Überschriften:

```
h1, h2 {
  text-wrap: balance;
}
```

Wenn du möchtest, kann ich dir im nächsten Schritt noch eine **kompakte Cheat Sheet-Tabelle**, eine **Browser-Kompatibilitätsübersicht** oder eine **visuelle Demo mit HTML/CSS-Beispielen für jeden Wert** erstellen.

SASS Mixin für typografische Kontrolle in Bricks Builder

Ich nutze den BricksBuilder und dort kann ich dank SASS (durch das Plugin Advanced Themer) Mixins verwenden. Also dort gibt es ja folgende Syntax:

```
%root% {
color: red;
}
```

Kann du mir ein Mixin schreiben, das ich einfach in beliebige Klassen einfügen kann und das folgendes macht?:

Es soll 1. einen Parameter entgegen nehmen (1, 2 oder 3):

1. Wörter brechen typografisch korrekt um, so wie es in der gerade aktiven Sprache (auf der Website) korrekt ist. D.h. also "Mund-schenk" beispielsweise (im Deutschen).
2. Es finden keine Umbrüche mitten in Wörtern statt (nur ganze Wörter), außer die gesamte Zeile reicht für ein einziges Wort nicht aus, dann soll es wie bei 1. umbrochen werden.
3. Umbrüche finden gar nicht mehr automatisch statt. Nur manuelle Umbrüche (z.B. `<br>`) funktionieren.

Dann soll das Mixing auch noch unabhängig vom gewählten Parameter-Wert festlegen, dass spracheabhängige typografische Zeichen gesetzt werden: Beispielsweise im Deutschen statt `""` dann `„“`, statt `''` dann `‚‘`, statt `--` dann `–`, usw.

Diese Regeln sollen aber intelligent gesetzt werden, d.h. z.B. nicht in `pre`-Elementen usw.

Hier ist ein umfassendes Mixin für deine Anforderungen:

```
// ============================================
// TYPOGRAFIE MIXIN
// ============================================
// Parameter:
// 1 = Silbentrennung (sprachabhängig korrekt)
// 2 = Nur ganze Wörter (Fallback auf Silbentrennung)
// 3 = Keine automatischen Umbrüche
```

```
// ============================================

@mixin typo-wrap($mode: 2) {
  // ========================================
  // GEMEINSAME EINSTELLUNGEN (immer aktiv)
  // ========================================

  // Sprache für Anführungszeichen und Trennung erkennen
  // Browser nutzt das lang-Attribut des HTML-Elements

  // Typografische Anführungszeichen aktivieren
  // Deutsch: „" und ,'
  // Englisch: "" und ''
  // Französisch: «» und ‹›
  // etc. - Browser wählt automatisch basierend auf lang=""
  quotes: auto;

  // Typografische Optimierungen
  font-variant-numeric: oldstyle-nums; // Optional: Mediävalziffern
  font-kerning: auto;
  font-feature-settings: "kern" 1, "liga" 1, "calt" 1;

  // Witwen und Waisen vermeiden (moderne Browser)
  text-wrap: pretty;

  // ========================================
  // UMBRUCH-MODI
  // ========================================

  @if $mode == 1 {
    // MODUS 1: Silbentrennung (sprachabhängig)
    // Wörter werden an korrekten Silbengrenzen getrennt
    hyphens: auto;
    -webkit-hyphens: auto;
    -ms-hyphens: auto;
    hyphenate-limit-chars: 6 3 2; // Min. Wortlänge, vor Trennung, nach Trennung
    hyphenate-limit-lines: 2; // Max. aufeinanderfolgende getrennte Zeilen
    hyphenate-limit-zone: 8%; // Zone am Zeilenende für Trennung
    word-wrap: break-word;
```

```
  overflow-wrap: break-word;
  word-break: normal;

} @else if $mode == 2 {
  // MODUS 2: Ganze Wörter bevorzugen
  // Umbruch nur zwischen Wörtern, Silbentrennung nur als Fallback
  hyphens: auto;
  -webkit-hyphens: auto;
  -ms-hyphens: auto;
  overflow-wrap: break-word;
  word-wrap: break-word;
  word-break: normal;
  // Strengere Trennregeln
  hyphenate-limit-chars: 12 6 4; // Nur lange Wörter trennen
  hyphenate-limit-lines: 1;
  hyphenate-limit-zone: 20%; // Größere Zone = weniger Trennungen

} @else if $mode == 3 {
  // MODUS 3: Keine automatischen Umbrüche
  // Nur manuelle Umbrüche (<br>, \A) funktionieren
  hyphens: none;
  -webkit-hyphens: none;
  -ms-hyphens: none;
  white-space: normal;
  word-break: keep-all;
  overflow-wrap: normal;
  word-wrap: normal;
}

// ========================================
// AUSNAHMEN für Code-Elemente
// ========================================

pre,
code,
kbd,
samp,
var,
tt,
```

```
  [class*="code"],
  [class*="pre"],
  .wp-block-code,
  .wp-block-preformatted {
    // Keine typografischen Ersetzungen in Code
    quotes: none;
    hyphens: none;
    -webkit-hyphens: none;
    font-feature-settings: normal;
    font-variant-numeric: normal;
    // Code darf umbrechen wenn nötig
    white-space: pre-wrap;
    word-break: break-all;
    overflow-wrap: break-word;
  }
}

// ============================================
// ZUSATZ: Typografische Zeichen per CSS
// ============================================
// Für echte Zeichenersetzung (-- → –) benötigst du
// serverseitiges Processing (PHP/JS), aber CSS kann
// helfen mit Anführungszeichen:

// ============================================
// SHORTHAND MIXINS
// ============================================

@mixin typo-hyphenate {
  @include typo-wrap(1);
}

@mixin typo-whole-words {
  @include typo-wrap(2);
}

@mixin typo-no-break {
  @include typo-wrap(3);
}
```

Anwendung in Bricks Builder

```
// Beispiel 1: Fließtext mit Silbentrennung
%root% {
  @include typo-wrap(1);
}

// Beispiel 2: Überschriften - ganze Wörter
%root% {
  @include typo-wrap(2);
}

// Beispiel 3: Buttons/Labels - keine Umbrüche
%root% {
  @include typo-wrap(3);
}

// Oder mit Shorthand:
%root% {
  @include typo-hyphenate;    // = Modus 1
  @include typo-whole-words;  // = Modus 2
  @include typo-no-break;     // = Modus 3
}
```

Wichtige Hinweise

1. HTML-Sprachattribut setzen

Damit die automatische Silbentrennung funktioniert, muss das `lang`-Attribut gesetzt sein:

```
<html lang="de">  <!-- Deutsch -->
<html lang="en">  <!-- Englisch -->
<html lang="de-CH">  <!-- Schweizerdeutsch -->
```

2. Gedankenstriche (-- → –)

CSS kann **keine Textzeichen ersetzen**. Für automatische Ersetzung von `--` zu `–` benötigst du:

Option A: JavaScript (clientseitig)

```
// In Bricks unter "Page Settings > Custom Code > Body Scripts"
document.querySelectorAll('p, h1, h2, h3, h4, h5, h6, li, td, th, blockquote')
  .forEach(el => {
    if (!el.closest('pre, code, script, style')) {
      el.innerHTML = el.innerHTML
        .replace(/--/g, '–')
        .replace(/\.\.\./g, '…');
    }
  });
```

Option B: WordPress-Filter (serverseitig, empfohlen)

```
// In functions.php oder Code-Snippets Plugin
add_filter('the_content', 'typografie_korrektur', 99);
add_filter('the_title', 'typografie_korrektur', 99);

function typografie_korrektur($content) {
    $replacements = [
        '--' => '–',      // Gedankenstrich
        '---' => '–',     // Geviertstrich
        '...' => '…',     // Auslassungspunkte
        '(c)' => '©',
        '(r)' => '®',
        '(tm)' => '™',
    ];
    return str_replace(
        array_keys($replacements),
        array_values($replacements),
        $content
    );
}
```

3. Browser-Support

Feature	Chrome	Firefox	Safari	Edge

`hyphens: auto`	✅	✅	✅	✅
`quotes: auto`	✅	✅	✅	✅
`hyphenate-limit-*`	⚠	✅	✅	⚠
`text-wrap: pretty`	✅	✅	✅	✅

Die Basis-Funktionalität wird von allen modernen Browsern unterstützt!

OKLCH

HSL-L ist nicht „wahrgenommene Helligkeit“ □□

Der `L`-Wert in **HSL** ist nur eine geometrische Größe innerhalb dieses Farbmodells, aber **nicht** so definiert, dass Menschen Farben bei gleichem `L` als gleich hell wahrnehmen.

Ein typisches Beispiel:

- ein kräftiges **Gelb**
- ein kräftiges **Blau**

können in HSL denselben `L`-Wert haben, aber Gelb wirkt trotzdem viel heller als Blau.

Warum ist das so?

HSL basiert letztlich auf dem üblichen **RGB-Farbraum**. Dieser ist praktisch für Bildschirme, aber nicht *wahrnehmungslinear*.

Das heißt:

- gleiche numerische Änderungen in RGB oder HSL
- führen **nicht** zu gleich großen visuellen Änderungen

Insbesondere ist unser Auge für verschiedene Farbbereiche unterschiedlich empfindlich:

- **Gelb/Grün** wirkt oft sehr hell
- **Blau** wirkt oft deutlich dunkler
- obwohl die technischen Werte „ähnlich“ aussehen

Was man stattdessen braucht

Wenn du Farben *wirklich gleich hell* machen willst, brauchst du einen **perzeptuell gleichmäßigeren Farbraum**. Dafür gibt es mathematische Modelle.

Die wichtigsten sind:

1. **CIELAB / L*a*b***
   - `L*` steht näher an der wahrgenommenen Helligkeit
   - deutlich besser als HSL
2. **CIELUV**
   - ähnlicher Zweck, etwas anderer Schwerpunkt
3. **OKLab / OKLCH**
   - moderner
   - für UI, Web und Design oft sehr gut geeignet
   - in vielen Fällen heute die praktisch beste Wahl

Die eigentliche Idee: „Luminanz“ und „perzeptuelle Helligkeit“

Es gibt dabei zwei verwandte, aber verschiedene Dinge:

1. Physikalische bzw. technische Helligkeit: relative Luminanz

Für einen sRGB-Farbwert berechnet man zunächst die **relative Luminanz** Y.

Wenn `R`, `G`, `B` im Bereich 0 bis 1 als **lineare** RGB-Werte vorliegen, dann gilt:

$$
Y = 0.2126 R + 0.7152 G + 0.0722 B
$$

Wichtig: Das sind **lineare** RGB-Werte, nicht die direkt aus CSS bekannten 8-Bit-sRGB-Werte.

Vorher muss man sRGB erst linearisieren. Für einen sRGB-Kanal c_{srgb} gilt:

$$
c_{lin} =
\begin{cases}
\frac{c_{srgb}}{12.92}, & c_{srgb} \le 0.04045 \
\left(\frac{c_{srgb}+0.055}{1.055}\right)^{2.4}, & c_{srgb} > 0.04045
\end{cases}
$$

Diese Luminanz ist z. B. wichtig für:

- **Kontrastberechnungen**
- WCAG
- technische Lichtstärke-Vergleiche

Aber: gleiche Luminanz ist noch nicht perfekt dasselbe wie „gleich hell empfunden“.

---

2. Wahrgenommene Helligkeit: perzeptuelle Lightness

Dafür gibt es z. B. in **CIELAB** die Größe L^*.

Aus der relativen Luminanz Y und dem Referenz-Weiß Y_n wird näherungsweise berechnet:

$$
L^* = 116 \cdot \left(\frac{Y}{Y_n}\right)^{1/3} - 16
$$

für größere Werte; genauer ist die Definition stückweise, aber die Grundidee ist:

- nicht linear,
- sondern an die menschliche Wahrnehmung angepasst.

Gleiche L^*-Werte sind viel näher an „gleich hell“ als gleiche HSL-`L`-Werte.

Für CSS heute besonders interessant: OKLCH

Für praktische Arbeit im Web ist **OKLCH** oft die beste Antwort.

OKLCH besteht aus:

- **L** = wahrgenommene Helligkeit
- **C** = Chroma / Farbstärke
- **H** = Farbwinkel

Das ist also ähnlich intuitiv wie HSL, aber viel besser an die Wahrnehmung angepasst.

Vorteil

Wenn du in OKLCH den **gleichen `L`-Wert** verwendest, wirken Farben wesentlich eher gleich hell.

Beispiel in CSS:

```
color: oklch(0.7 0.15 30);
color: oklch(0.7 0.15 120);
color: oklch(0.7 0.15 260);
```

Hier ist die Helligkeit durch `0.7` deutlich konsistenter als bei vergleichbaren HSL-Farben.

Also: Gibt es eine mathematische Rechenweise?

Ja. Sogar mehrere — je nachdem, was du mit „hell“ meinst:

1. **Gleiche technische Luminanz**
   - über lineares RGB und
     $$
     Y = 0.2126 R + 0.7152 G + 0.0722 B
     $$
   - gut für Kontrast und Messbarkeit
2. **Gleiche wahrgenommene Helligkeit**
   - über **CIELAB** `L*` oder besser **OKLab/OKLCH** `L`
   - gut für Design, UI und Farbpaletten

Praktische Empfehlung

Wenn dein Ziel ist:

„Farben sollen im Interface gleich hell wirken“

Dann nimm **OKLCH** statt HSL.

„Ich will Kontraste mathematisch korrekt prüfen“

Dann arbeite mit **relativer Luminanz**.

„Ich will ein älteres, etabliertes Farbmodell“

Dann ist **CIELAB** eine gute Wahl.

Warum HSL trotzdem oft benutzt wird

HSL ist beliebt, weil es sehr intuitiv aussieht:

- `H` = Farbton
- `S` = Sättigung
- `L` = hell/dunkel

Aber dieses `L` ist eher ein *Modellparameter* als echte visuelle Helligkeit.

Man kann HSL gut verwenden für:

- einfache Farbvarianten
- schnelle Prototypen
- kleine Anpassungen

Aber nicht, wenn du exakt gleich helle Farben willst.

Kurzfassung

Ja, dafür gibt es mathematische Modelle.
Wenn Farben **wirklich gleich hell wirken** sollen, ist **HSL ungeeignet**.

Am besten nimmst du:

- **OKLCH** für praktische CSS-/UI-Arbeit
- oder **CIELAB / OKLab**, wenn du mathematisch sauber in perzeptuellen Farbräumen arbeiten willst

Wenn du möchtest, kann ich dir als Nächstes noch zeigen:

1. **warum HSL-L mathematisch so definiert ist**,
2. **wie man eine HSL-Farbe in relative Luminanz umrechnet**, oder
3. **wie man in CSS mit OKLCH eine gleichhelle Farbpalette baut**.

OKLCH - SCSS/SASS - Dark/Light Mode

Wenn deine Farben bereits als `oklch(...)` vorliegen, kannst du eine Funktion bauen, die einfach den **Lightness-Wert umkehrt**:

$$
L_{\text{neu}} = 1 - L
$$

Also aus:

```
oklch(0.3 0.15 30)
```

wird:

```
oklch(0.7 0.15 30)
```

Einfache Sass-Funktion

```
@use "sass:color";
@use "sass:math";

@function invert-oklch-lightness($c) {
  $space: color.space($c);

  @if $space != oklch {
    @error "invert-oklch-lightness erwartet eine OKLCH-Farbe, bekommen: #{$space}";
  }

  $l: color.channel($c, "lightness", $space: oklch);
  $chroma: color.channel($c, "chroma", $space: oklch);
```

```
  $hue: color.channel($c, "hue", $space: oklch);

  $new-l: 1 - $l;

  @return oklch($new-l $chroma $hue);
}
```

Verwendung

```
$color: oklch(0.3 0.15 30);
$dark-mode: invert-oklch-lightness($color);
```

Ergebnis:

```
oklch(0.7 0.15 30)
```

Beispiel im Einsatz

```
@use "sass:color";

@function invert-oklch-lightness($c) {
  @if color.space($c) != oklch {
    @error "Nur OKLCH-Farben sind erlaubt.";
  }

  $l: color.channel($c, "lightness", $space: oklch);
  $chroma: color.channel($c, "chroma", $space: oklch);
  $hue: color.channel($c, "hue", $space: oklch);

  @return oklch(1 - $l $chroma $hue);
}

$bg-light: oklch(0.92 0.03 240);
$bg-dark: invert-oklch-lightness($bg-light);

$accent-light: oklch(0.3 0.15 30);
```

```
$accent-dark: invert-oklch-lightness($accent-light);

:root {
  --bg: #{$bg-light};
  --accent: #{$accent-light};
}

@media (prefers-color-scheme: dark) {
  :root {
    --bg: #{$bg-dark};
    --accent: #{$accent-dark};
  }
}
```

Wichtiger Hinweis

Das ist **mathematisch simpel**, aber **gestalterisch nicht immer optimal** □

Denn ein „guter“ Dark-Mode ist oft **nicht nur**:

1. Lightness invertieren
2. Chroma gleich lassen
3. Hue gleich lassen

Sondern häufig will man zusätzlich:

- **Chroma etwas reduzieren**, weil kräftige Farben auf dunklem Hintergrund schnell zu intensiv wirken
- bei sehr hellen Farben ein anderes Mapping verwenden
- Kontraste für Text gezielt anpassen

Etwas robustere Variante

Wenn du willst, kannst du beim Invertieren gleichzeitig die Chroma leicht dämpfen:

```
@use "sass:color";
@use "sass:math";
```

```
@function dark-mode-oklch($c, $chroma-factor: 0.85) {
  @if color.space($c) != oklch {
    @error "Nur OKLCH-Farben sind erlaubt.";
  }

  $l: color.channel($c, "lightness", $space: oklch);
  $chroma: color.channel($c, "chroma", $space: oklch);
  $hue: color.channel($c, "hue", $space: oklch);

  $new-l: 1 - $l;
  $new-chroma: $chroma * $chroma-factor;

  @return oklch($new-l $new-chroma $hue);
}
```

Beispiel:

```
$color: oklch(0.3 0.15 30);
$dark: dark-mode-oklch($color);
```

Ergibt ungefähr:

```
oklch(0.7 0.1275 30)
```

Falls du nur rohe Werte übergeben willst

Wenn du lieber direkt mit Zahlen statt mit einem Farbwert arbeitest:

```
@function invert-oklch($l, $c, $h) {
  @return oklch(1 - $l $c $h);
}
```

Verwendung:

```
$color: invert-oklch(0.3, 0.15, 30);
```

Fazit

Ja, absolut.
Für deinen konkreten Fall reicht im Prinzip genau das:

```
@function invert-oklch-lightness($c) {
  $l: color.channel($c, "lightness", $space: oklch);
  $chroma: color.channel($c, "chroma", $space: oklch);
  $hue: color.channel($c, "hue", $space: oklch);

  @return oklch(1 - $l $chroma $hue);
}
```

Wenn du möchtest, kann ich dir als Nächstes auch noch eine Version schreiben, die:

1. **auch normale CSS-Strings wie `oklch(0.3 0.15 30)` verarbeitet**, oder
2. **eine ganze Sass-Map mit Theme-Farben automatisch in Dark-Mode-Werte umwandelt**.

HSL lightness is not true perceived brightness □□

The `L` value in **HSL** is only a geometric quantity within that color model. It is **not** defined so that colors with the same `L` are perceived by humans as equally bright.

A typical example:

1. a strong **yellow**
2. a strong **blue**

can have the same HSL `L` value, yet yellow will still appear much brighter than blue.

Why is that?

HSL is ultimately based on the usual **RGB color space**. RGB is practical for displays, but it is not *perceptually uniform*.

That means:

1. equal numerical changes in RGB or HSL
2. do **not** produce equally large visual changes

In particular, our eyes are not equally sensitive across all hues:

1. **yellow/green** often appears very bright
2. **blue** often appears much darker
3. even when the technical values look similar

What you need instead

If you want colors to be *truly equally bright*, you need a **more perceptually uniform color space**. There are mathematical models for that.

The most important ones are:

1. **CIELAB / L*a*b***
    1. `L*` is much closer to perceived lightness
    2. much better than HSL
2. **CIELUV**
    1. similar purpose
    2. slightly different focus
3. **OKLab / OKLCH**
    1. more modern
    2. often very well suited for UI, web, and design work
    3. in many cases, the best practical choice today

The core idea: “luminance” vs. “perceived lightness”

These are related, but not the same thing.

1. Physical or technical brightness: relative luminance

For an sRGB color, you can first compute the **relative luminance** Y.

If `R`, `G`, and `B` are **linear** RGB values in the range from 0 to 1, then:

$$
Y = 0.2126 R + 0.7152 G + 0.0722 B
$$

Important: these are **linear** RGB values, not the raw 8-bit sRGB values you typically use in CSS.

So first, sRGB must be linearized. For an sRGB channel c_{srgb}:

$$
c_{lin} =
\begin{cases}
\frac{c_{srgb}}{12.92}, & c_{srgb} \le 0.04045 \\
\left(\frac{c_{srgb}+0.055}{1.055}\right)^{2.4}, & c_{srgb} > 0.04045
\end{cases}
$$

This luminance is important for things like:

1. **contrast calculations**
2. WCAG
3. technical comparisons of brightness

But equal luminance is still not perfectly the same as “equally bright as perceived.”

2. Perceived brightness: perceptual lightness

For this, **CIELAB** uses the quantity L^*.

From the relative luminance Y and the reference white Y_n, it is approximately calculated as:

$$
L^* = 116 \cdot \left(\frac{Y}{Y_n}\right)^{1/3} - 16
$$

for larger values; more precisely, the full definition is piecewise. But the main idea is:

1. it is not linear
2. it is adjusted to human perception

Equal L^* values are much closer to “equally bright” than equal HSL `L` values.

Especially relevant for CSS today: OKLCH

For practical work on the web, **OKLCH** is often the best answer.

OKLCH consists of:

1. **L** = perceived lightness
2. **C** = chroma / colorfulness
3. **H** = hue angle

So it is similar to HSL in terms of intuition, but much better aligned with perception.

Advantage

If you use the same `L` value in OKLCH, colors are much more likely to appear equally bright.

Example in CSS:

```
color: oklch(0.7 0.15 30);
color: oklch(0.7 0.15 120);
color: oklch(0.7 0.15 260);
```

Here, the lightness set by `0.7` is much more consistent than in comparable HSL colors.

So: is there a mathematical way to do this?

Yes. In fact, there are several—depending on what exactly you mean by “equally bright”:

1. **Equal technical luminance**
   1. using linear RGB and
      $$
      Y = 0.2126 R + 0.7152 G + 0.0722 B
      $$
   2. good for contrast and measurable brightness
2. **Equal perceived lightness**
   1. using **CIELAB** `L*` or, better, **OKLab/OKLCH** `L`
   2. good for design, UI, and color palettes

Practical recommendation

If your goal is:

“Colors should appear equally bright in an interface”

Use **OKLCH** instead of HSL.

“I want mathematically correct contrast checks”

Work with **relative luminance**.

“I want an older, well-established color model”

Then **CIELAB** is a good choice.

Why HSL is still often used

HSL is popular because it feels intuitive:

1. `H` = hue
2. `S` = saturation
3. `L` = light/dark

But that `L` is more of a *model parameter* than a true measure of visual lightness.

HSL can still be useful for:

1. simple color variations
2. quick prototypes
3. small adjustments

But not if you want colors that are precisely equal in perceived brightness.

Short version

Yes, there are mathematical models for this.
If colors should **really appear equally bright**, **HSL is not suitable**.

The best options are:

1. **OKLCH** for practical CSS and UI work

2. **CIELAB / OKLab** if you want to work in perceptually meaningful color spaces more rigorously

If you want, I can also show you next:

1. **why HSL lightness is defined the way it is**
2. **how to convert an HSL color into relative luminance**
3. **how to build an equal-lightness color palette in CSS using OKLCH**

OKLCH

Ways to define colors in CSS — from HEX to OKLCH □□

CSS has accumulated quite a few ways to describe color over the years. Some are familiar and compact, like `#ff0000`; some are practical and expressive, like `rgb(255 0 0 / 0.5)`; and some are modern, powerful, and much better aligned with human perception—especially **`oklch()`**.

If you write CSS today, it's worth understanding not just *how* these formats work, but *when* each one makes sense. This article walks through the main color syntaxes in CSS, explains their strengths and trade-offs, and gives special attention to **OKLCH**, which is becoming one of the most useful ways to work with color in modern design systems.

Why CSS has so many color formats

Different color formats exist because they solve different problems:

1. **Convenience**
    1. Named colors like `red` are easy to remember.
    2. HEX is compact and common in design tools.
2. **Control**
    1. `rgb()` and `hsl()` make it easier to think in channels.
    2. Alpha transparency can be expressed clearly.
3. **Color science**
    1. Newer spaces like `lab()`, `lch()`, `oklab()`, and `oklch()` aim to make color adjustments more perceptually meaningful.
    2. They help produce more consistent palettes, gradients, and theme systems.

In short: older formats are still useful, but newer ones are often better for design work that needs consistency and nuance.

Named colors

CSS supports a long list of predefined color keywords:

```
color: red;
background: rebeccapurple;
border-color: lightgray;
```

These are readable and quick, but they have limits:

1. They cover only a fixed set of colors.
2. They aren’t precise enough for most branding or system design.
3. Their perceived brightness and saturation vary widely.

Named colors are nice for demos, quick experiments, and a few memorable values like `white`, `black`, `transparent`, or `rebeccapurple`. For serious visual design, developers usually move to numeric color functions.

HEX colors

HEX is one of the most recognizable CSS color formats:

```
color: #ff0000;
color: #f00;
```

These two values mean the same thing: pure red.

How HEX works

A 6-digit HEX color is structured like this:

```
#RRGGBB
```

Each pair is a channel from `00` to `ff`:

1. `RR` = red
2. `GG` = green
3. `BB` = blue

So:

```
#ff0000 /* red */
#00ff00 /* green */
```

```
#0000ff /* blue */
#ffffff /* white */
#000000 /* black */
```

There is also shorthand:

```
#f00   /* equivalent to #ff0000 */
#0f0   /* equivalent to #00ff00 */
#fff   /* equivalent to #ffffff */
```

And alpha can be included too:

```
#ff000080 /* red at about 50% opacity */
#f008     /* shorthand with alpha */
```

When HEX is useful

HEX is popular because it is:

1. **Compact**
2. **Widely recognized**
3. **Easy to copy from design tools**

But it also has some drawbacks:

1. It’s not very intuitive for humans.
    1. Looking at `#7a5ccf` doesn’t tell you much immediately.
2. Adjusting colors manually is awkward.
    1. Making something “a bit lighter” is not straightforward.
3. It reflects RGB encoding rather than human perception.
    1. Equal numeric changes do not feel like equal visual changes.

HEX is still common and totally valid, but it’s often best thought of as a storage or interchange format rather than the most ergonomic design format.

RGB and RGBA

RGB expresses colors through red, green, and blue channels:

```
color: rgb(255 0 0);
color: rgb(255, 0, 0);
```

Both forms are understood in CSS, though the space-separated modern syntax is generally preferred.

You can also include alpha:

```
color: rgb(255 0 0 / 50%);
color: rgb(255 0 0 / 0.5);
```

Why RGB is useful

RGB maps directly to how screens emit light, so it is fundamental in digital color. It's often useful when:

1. You need direct channel control.
2. You want explicit alpha handling.
3. You are working with JavaScript, canvas, or generated colors.

Example:

```
background-color: rgb(34 197 94 / 0.2);
```

That said, RGB has the same conceptual limitation as HEX: it is not a perceptual color space. Two colors with similar channel differences may not *look* similarly different.

HSL and HSLA

HSL stands for **Hue, Saturation, Lightness**:

```
color: hsl(0 100% 50%);
color: hsl(240 100% 50%);
color: hsl(120 100% 25% / 0.8);
```

Many developers like HSL because it feels more intuitive than RGB.

What the parts mean

1. **Hue**
    1. The basic position on the color wheel, usually in degrees.
    2. `0` is red, `120` is green, `240` is blue.
2. **Saturation**

    1. How vivid or grayish the color is.
    2. `0%` is gray, higher values are more colorful.
3. **Lightness**
    1. How light or dark the color appears.
    2. `0%` is black, `100%` is white.

Why HSL became popular

HSL is often easier to reason about when making quick adjustments:

```
/* same hue, different lightness */
--blue-40: hsl(220 80% 40%);
--blue-50: hsl(220 80% 50%);
--blue-60: hsl(220 80% 60%);
```

This *looks* appealing, but there is a catch: **HSL lightness is not perceptually uniform**. Colors with the same HSL lightness often do not appear equally bright. Yellow, for example, tends to look much brighter than blue at the same nominal lightness.

So HSL is intuitive, but it can be misleading when building balanced palettes.

HWB

CSS also supports `hwb()`, which means **Hue, Whiteness, Blackness**:

```
color: hwb(200 20% 10%);
```

This format is less common, but it can feel natural because it describes a hue mixed with white and black. It is useful in certain tooling and algorithmic color generation, though it has not become as mainstream as HEX, RGB, or HSL.

For many authors, `hwb()` is more of a nice-to-know syntax than an everyday tool.

Lab and LCH

As CSS evolved, it began to support more perceptually oriented color spaces.

`lab()`

`lab()` is based on the CIE Lab color space:

```
color: lab(60% 30 20);
```

It uses:

1. **L** for lightness
2. **a** for the green–red axis
3. **b** for the blue–yellow axis

`lch()`

`lch()` is a cylindrical representation of Lab:

```
color: lch(60% 50 30);
```

It uses:

1. **L** for lightness
2. **C** for chroma
3. **H** for hue

This already feels more designer-friendly than `lab()`, because **chroma** and **hue** are easier to reason about than `a` and `b`.

Lab and LCH were important steps toward more perceptual CSS color handling, but in practice, **OKLab** and **OKLCH** are often even better choices.

OKLab and OKLCH — the modern highlight

If there is one color format worth learning now, it is **`oklch()`**.

Why OKLCH matters

OKLCH is built on **OKLab**, a color space designed to be more perceptually uniform than older RGB/HSL-style approaches and often more practical than CIE Lab/LCH for interface and web design.

The big promise is simple:

> If you change a value in OKLCH, the visual result is more likely to match what you *expected*.

This is incredibly helpful for:

1. Building color scales
2. Creating themes
3. Keeping brightness consistent across hues
4. Producing smoother gradients
5. Making systematic adjustments with less guesswork

The structure of `oklch()`

An OKLCH color looks like this:

```
color: oklch(62% 0.18 264);
```

The three main components are:

1. **Lightness**
   1. Controls how light or dark the color appears.
   2. Usually the easiest value to tune for contrast and hierarchy.
2. **Chroma**
   1. Controls color intensity.
   2. Roughly similar to saturation, but more grounded in the color space.
   3. Higher values are more vivid, though the achievable maximum depends on hue.
3. **Hue**
   1. Controls the color family.
   2. Expressed as an angle.

Alpha can be added too:

```
color: oklch(62% 0.18 264 / 0.7);
```

Why designers and developers like OKLCH

1. Lightness behaves more sensibly

With HSL, equal lightness values across different hues often look inconsistent. In OKLCH, lightness is much closer to perceived lightness.

That means this kind of palette is more reliable:

```
--purple-40: oklch(40% 0.16 300);
--purple-55: oklch(55% 0.16 300);
--purple-70: oklch(70% 0.16 300);
```

Changing the first number tends to produce a more predictable visual ramp.

2. Chroma is better than “saturation” for many tasks

“Saturation” in HSL can be deceptive. A color at `100%` saturation is not necessarily the most vivid or the most balanced version of that hue in practice.

In OKLCH, **chroma** is a more useful measure of colorfulness. If you reduce chroma, colors generally become more muted in a way that feels more natural:

```
--brand-strong: oklch(60% 0.22 250);
--brand-soft:   oklch(60% 0.10 250);
```

These are easier to think about as “same lightness, same hue, different intensity.”

3. Hue shifts are easier to manage

Because OKLCH is designed for perceptual consistency, rotating hue while holding other values steady often gives more balanced results than in older spaces.

This is valuable for generating semantic palettes:

```
--info:    oklch(65% 0.14 240);
--success: oklch(65% 0.14 145);
--warning: oklch(65% 0.14 85);
--danger:  oklch(65% 0.14 25);
```

These colors are not magically perfect, but they often start from a much better baseline.

Reading OKLCH intuitively

A useful way to think about it is:

```
oklch(lightness chroma hue)
```

For example:

```
oklch(70% 0.12 210)
```

can be read as:

1. A fairly light color
2. With moderate intensity
3. In the blue-cyan range

Over time, this becomes surprisingly ergonomic.

Practical examples

A button palette

```
:root {
  --button-bg: oklch(62% 0.17 260);
  --button-bg-hover: oklch(56% 0.17 260);
  --button-text: oklch(98% 0.01 260);
}
```

This works well because:

1. Hover is created mainly by lowering lightness.
2. Hue and chroma stay stable.
3. The relationship between states remains coherent.

A muted surface system

```
:root {
  --surface-1: oklch(99% 0.01 250);
  --surface-2: oklch(96% 0.01 250);
  --surface-3: oklch(92% 0.02 250);
  --text-1: oklch(22% 0.02 250);
  --text-2: oklch(38% 0.02 250);
}
```

Notice how even “neutral” grays can carry a slight hue bias. That can make an interface feel warmer, cooler, softer, or more branded without becoming obviously colored.

Important caveat: gamut limits

OKLCH is excellent, but not every combination of lightness, chroma, and hue can be displayed on a typical screen. In other words, some values fall **outside the available gamut**.

This matters because:

1. Very high chroma may not be possible for certain hues.
2. The browser may clamp or adjust out-of-range values.
3. A color that is mathematically valid may not render as expected on all devices.

So while OKLCH is powerful, it still has real-world limits. A practical workflow is to:

1. Start with reasonable chroma values
2. Preview in actual browsers
3. Use tooling that can warn about gamut issues

Alpha and transparency

Most modern CSS color functions support alpha using slash syntax:

```
rgb(0 0 0 / 0.5)
hsl(220 20% 20% / 0.3)
oklch(60% 0.15 240 / 0.4)
```

This is cleaner than older function pairs like `rgba()` and `hsla()`. While those names still exist historically, modern CSS treats alpha as a natural part of the same color function.

That consistency is one of the nice improvements in modern CSS syntax.

Choosing the right format

There is no single correct format for every situation. A sensible rule of thumb is:

1. **Named colors**
    1. Good for simple cases and quick demos
2. **HEX**
    1. Good for compatibility, compactness, and copy-paste from design tools
3. **RGB**
    1. Good when you want direct channel control or explicit programmatic handling
4. **HSL**
    1. Good for quick human-readable adjustments
    2. Less reliable for perceptual consistency

5. **Lab/LCH**
    1. Good for more advanced perceptual workflows
    2. Often overshadowed by OKLab/OKLCH for modern UI work
6. **OKLCH**
    1. Excellent for design systems, palettes, theming, and more visually consistent adjustments
    2. Often the best modern choice when browser support fits your audience

A few practical recommendations

If you're building modern CSS today, here's a pragmatic approach:

1. Use **OKLCH** when you are defining a system of colors.
    1. Especially for scales, semantic tokens, interactive states, and theme variants.
2. Use **HEX or RGB** when integration or tooling makes them more convenient.
    1. For example, some APIs, older docs, or design exports may still rely on them.
3. Be cautious with **HSL** for palette construction.
    1. It is intuitive, but its "lightness" can mislead you.
4. Always test actual contrast and rendering.
    1. No color space replaces accessibility checks.
    2. Perceptual consistency does not automatically guarantee sufficient contrast.

Final thoughts

CSS color has evolved from a handful of convenient notations into a genuinely rich system. HEX, RGB, and HSL are still useful and widely relevant, but they reflect older ways of thinking about color—more tied to encoding or simplified mental models than to human perception.

OKLCH stands out because it bridges technical precision and design intuition. It gives you a way to adjust lightness, intensity, and hue in a manner that much more closely matches what your eyes expect to see. For modern UI work, that makes it one of the most compelling color formats CSS has ever offered.

If you only take one thing away, let it be this: **learn** `oklch()`. Even if you continue using HEX or RGB in some places, understanding OKLCH will make you better at choosing, adjusting, and organizing color across the board.

Dark Mode

Dark Mode: Gängige Methoden

1. Über `prefers-color-scheme` automatisch dem System folgen

Der Browser erkennt, ob das Betriebssystem auf Hell oder Dunkel gestellt ist.

Beispiel

```
body {
  background: white;
  color: black;
}

@media (prefers-color-scheme: dark) {
  body {
    background: #121212;
    color: white;
  }
}
```

Vorteile

- Sehr einfach
- Kein JavaScript nötig
- Nutzt die Nutzerpräferenz automatisch

Nachteile

- Nutzer kann auf deiner Seite nicht unbedingt separat umschalten
- Weniger flexibel

---

2. Per CSS-Klasse auf `html` oder `body`

Du setzt z. B. eine Klasse wie `dark` auf das `<html>`- oder `<body>`-Element und definierst dafür eigene Styles.

Beispiel

```
<html class="dark">
```

```
body {
  background: white;
  color: black;
}

.dark body {
  background: #121212;
  color: white;
}
```

Vorteile

- Sehr verbreitet
- Einfach per JavaScript umschaltbar
- Gut mit Frameworks kombinierbar

Nachteile

- Man braucht etwas JS für den Toggle
- Bei vielen Komponenten kann das CSS unübersichtlich werden

---

3. Mit CSS Custom Properties (Variablen)

Oft die sauberste Lösung: Du definierst Farbvariablen und überschreibst sie im Dark Mode.

Beispiel

```
:root {
  --bg: white;
  --text: black;
}

.dark {
  --bg: #121212;
  --text: white;
}

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

Vorteile

- Sehr wartbar
- Farben zentral definiert
- Ideal für größere Projekte und Design-Systeme

Nachteile

- Anfangs etwas mehr Struktur nötig
- Man muss konsequent mit Variablen arbeiten

---

4. Kombination aus Systempräferenz + manuellem Toggle

Das ist in der Praxis oft die beste Lösung.

Typischer Ablauf

- Standardmäßig: `prefers-color-scheme` nutzen
- Optional: Nutzer kann selbst hell/dunkel wählen
- Auswahl in `localStorage` speichern
- Beim nächsten Besuch wiederherstellen

Beispielidee

- Wenn der Nutzer nichts gewählt hat → System folgen
- Wenn der Nutzer manuell gewählt hat → diese Wahl hat Vorrang

Vorteile

- Beste UX
- Flexibel
- Nutzer behält Kontrolle

Nachteile

- Etwas mehr Implementierungsaufwand
- Man sollte auf „Flash of wrong theme“ beim Laden achten

---

5. Framework-spezifische Lösungen

Viele Frameworks bringen eigene Patterns mit:

- **Tailwind CSS**: `dark:`-Klassen, z. B. `dark:bg-black`
- **Bootstrap**: teils über Data-Attribute oder Theme-Konfiguration
- **Material UI / Chakra UI / etc.**: Theme Provider, Light/Dark Tokens

Vorteil

- Schnell integrierbar
- Oft bereits gut dokumentiert

Nachteil

- Abhängig vom Framework
- Weniger „roh“ verständlich als pures CSS

Technisch gängige Umsetzungsmuster

A. Nur CSS

Gut für einfache Websites:

- `prefers-color-scheme`
- CSS-Variablen
- kein manueller Schalter

B. CSS + JavaScript Toggle

Gut für die meisten Websites:

- Theme-Klasse setzen (`dark`)
- Theme im `localStorage` speichern

- Optional Systempräferenz als Fallback

C. Design Tokens / Theming-System

Gut für große Anwendungen:

- Farben, Abstände, Komponenten über Tokens
- Light/Dark als Theme-Varianten
- oft mit Component Libraries

Worauf man achten sollte

1. Nicht nur Hintergrund und Text ändern

Auch wichtig:

- Buttons
- Links
- Borders
- Inputs
- Cards
- Modals
- Tabellen
- Code-Blöcke
- Icons
- Schatten

2. Kontrast

Dark Mode ist nicht einfach „schwarz mit weißem Text“.
Besser:

- leicht aufgehellte Dunkeltöne statt purem Schwarz
- nicht reinweißes Textweiß
- ausreichender Kontrast nach WCAG

3. Bilder und Logos

- Manche Logos funktionieren auf dunklem Hintergrund nicht
- Eventuell alternative Assets nutzen
- Bei Fotos sparsam mit Filtern sein

4. FOUC / Flash beim Laden vermeiden

Wenn das Theme erst nach dem Laden per JS gesetzt wird, sieht man kurz das falsche Theme.
Lösung:

- Theme sehr früh im `<head>` setzen
- gespeicherte Präferenz vor dem Rendern anwenden

5. Browser-UI anpassen

Mit

```
<meta name="color-scheme" content="light dark">
```

oder in CSS:

```
:root {
  color-scheme: light dark;
}
```

können native Form-Controls und Scrollbars besser zum Theme passen.

Empfehlung für die Praxis

Für moderne Websites ist meistens diese Kombination am sinnvollsten:

- **CSS Custom Properties** für Farben
- `prefers-color-scheme` als Standard
- **manueller Toggle** per JS
- **Speicherung in** `localStorage`

Das ist heute wahrscheinlich der gängigste und robusteste Ansatz.