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.

Revision #2
Created 2026-05-23 17:57:25 UTC by art10m
Updated 2026-05-23 18:02:39 UTC by art10m