Modern CSS

Master the Key Concepts of CSS for Modern Web Development

Second Edition

Joe Attardi

Apress®

Modern CSS

Master the Key Concepts of CSS for Modern Web Development

Second Edition

Joe Attardi

Apress®

Modern CSS: Master the Key Concepts of CSS for Modern Web Development, Second Edition

Joe Attardi
Billerica, MA, USA

ISBN-13 (pbk): 979-8-8688-1726-7 ISBN-13 (electronic): 979-8-8688-1727-4
https://doi.org/10.1007/979-8-8688-1727-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Anandadeep Roy
Editorial Assistant: Jessica Vakkili

Cover designed by eStudioCalamar

The cover image is from Pixabay.com

To Liz and Benjamin

Table of Contents

About the Author

Joe Attardi is a software engineer from the Boston area, specializing in front-end development. He has over 20 years of experience working with web technologies such as JavaScript, TypeScript, HTML, and CSS. He has built rich front-end experiences for companies such as Dell, Constant Contact, and Salesforce. He is the author of *Using Gatsby and Netlify CMS* (Apress, 2020) and *Web API Cookbook* (O'Reilly, 2024). You can find him on X at @JoeAttardi.

About the Technical Reviewer

Kevin Wilson is a seasoned computer industry professional with over 20 years of experience and a master's degree in computer science software engineering. His expertise covers computer programming, software development, IT support, computer networks, cybersecurity, web development, graphic design, digital photography, film production, and visual effects. As writer and director at Elluminet Press Ltd, he has authored numerous best-selling technology books and video training courses on topics including Microsoft Office, Windows, Mac, computer hardware, Python programming, and web development, used by learners worldwide. He is also an experienced lecturer and IT trainer, as well as a consultant and reviewer for various technical publications. Known for his clear, visual, step-by-step teaching style, Kevin excels at making complex subjects accessible to students, professionals, and everyday users alike.

Acknowledgments

First and foremost, thanks to my wonderful wife, Liz, for always supporting and believing in me (and dealing with my loud typing!). And my son, Benjamin, for giving me much needed breaks from writing for playtime.

Thanks to all my friends and family for always supporting and encouraging my interest in computers and technology.

Many thanks to the team at Apress, particularly Anandadeep Roy and Krishnan Sathyamurthy, for working with me on the second edition of *Modern CSS*.

I appreciate the technical feedback from Kevin Wilson.

And finally, my thanks to Louise Corrigan, who brought me on board with Apress for the first edition back in 2020.

Introduction

In this second edition of *Modern CSS*, we will again take a tour of modern CSS. Whether you're brand new to CSS or you have some experience and need a refresher, or if you want to catch up on the newest CSS techniques, this book will have something for you.

This book will *not* teach you color theory or good design techniques. The intent of this book is to give you a strong foundation with the various CSS technologies.

The second edition has been updated throughout to add additional content about newer CSS features, make some clarifications, and fix some mistakes. Here's what we'll cover:

In Chapter 1, we'll start at the very beginning and talk about what CSS is, how it works, and how stylesheets are structured. We'll look at the DOM, the CSSOM, and the render tree.

In Chapter 2, we will cover CSS selectors. These are critical to understand. Selectors determine what CSS styles are applied to what elements. We'll also explore the concept of specificity.

Once we've laid the groundwork, we'll start to talk about CSS concepts in Chapter 3 like the box model, units, colors, and overflow. We'll also look at CSS custom properties, better known as variables.

We'll finally start applying styles in Chapter 4, where we'll look at borders, box shadows, and opacity. We will see several ways to hide an element on the page.

In Chapter 5, we'll learn all about backgrounds and gradients (which are a type of background image).

Chapter 6 deals with the important topic of styling text. We'll learn about text styles and layout, as well as how to use web fonts.

We'll see how to lay out and position elements in Chapter 7. This covers the different positions such as static, relative, absolute, fixed, and sticky. Also, in this chapter, we'll see the topic of stacking contexts and Z-index, which often trip up even experienced developers (including the author!).

In Chapter 8, we'll cover CSS transforms. This allows you to apply transformations such as rotation, scale, and skew to elements.

Transforms can be combined with transitions, which is one topic of Chapter 9, to create all kinds of interesting effects. Transitions can be applied to transforms or a slew of other CSS properties. Chapter 9 also covers animations, which takes the concepts of transitions to the next level.

Chapter 10 is dedicated to the flexible box layout, or flexbox, which is a powerful one-dimensional layout tool that has excellent browser support. With flexbox, we can finally easily center a div!

Chapter 11 is all about CSS Grid, the latest and greatest layout CSS tool. We'll also take a look at the newer CSS Subgrid feature.

In Chapter 12, we'll explore the topic of responsive design. While it's not an exhaustive guide – entire books have been written on the subject – it lays a good foundation, covering topics such as media queries, container queries, and fluid typography.

Finally, Chapter 13 will cover some miscellaneous CSS topics that didn't fit elsewhere in the book.

CHAPTER 1

Introduction to CSS

CSS, or Cascading Style Sheets, is a language for applying styling and layout to HTML documents. You can use it for everything from changing text color to creating complex grid layouts to performing animated transitions and everything in between.

The interesting part of CSS is the “Cascading” part. Because more than one style rule can apply to a given HTML element, there needs to be some way to determine which rule should apply in the event of a conflict. The styles “cascade” from less specific to more specific selectors (a concept called *specificity*), and the most specific rule wins. If two rules have the same specificity, then whichever rule comes last in the stylesheet wins.

Anatomy of a CSS Rule

A CSS stylesheet consists of rules. CSS rules target HTML elements by using selectors that describe the elements that should be styled. As you’ll see later, elements can be selected in many ways.

Rule Syntax

A CSS rule consists of a selector followed by a collection of CSS properties, contained inside curly braces. The properties consist of a name and value, separated by a colon, and are separated from each other with semicolons. A property may have a single value or a collection of multiple values, depending on the property.

The properties in a rule are applied to every element in the document that matches the selector. Figure 1-1 shows an example of a CSS rule.

J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_1

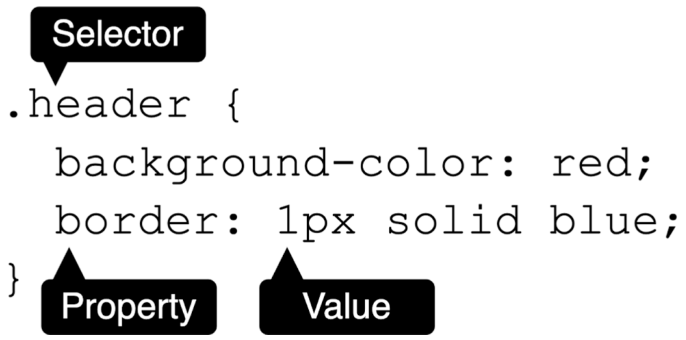


Figure 1-1. *The structure of a CSS rule*

This rule targets any element with the class `header` (more on classes later). Any element with this class will have a red background and a 1-pixel, solid, blue border. In this example, `background-color` and `border` are CSS properties.

Here, the background color is specified as `red`, but as you'll see later, there are many ways to represent colors in CSS. The border width is specified using the `px` unit, which corresponds to pixels. There are many other units including `em`, `rem`, and `%`. We'll cover some of the different CSS units in Chapter 3.

Property Conflicts

If the same property is used more than once in a rule, the last style in the rule is applied. In Listing 1-1, the element with the class `header` will have a blue background. The second `background-color` property overrides the first.

Listing 1-1. A CSS rule with conflicting properties

```css
.header {
  background-color: red;
  background-color: blue;
}
```

Similarly, if there are conflicting styles across multiple rules, the conflicting styles in the last rule are applied.

Listing 1-2. CSS rules with conflicting properties

```
.header {
  background-color: red;
}

.header {
  background-color: blue;
}
```

In Listing 1-2, the element with the class `header` will again have a blue background because the last rule in the stylesheet has a `background-color` of `blue`.

Comments

CSS can also contain comments, inside and outside of rules. Listing 1-3 shows some examples of CSS comments.

Listing 1-3. A CSS stylesheet with comments

```
/* This is a comment outside of a rule */

.header {
  /* This is a comment inside of a rule */
  background-color: red;
}
```

At-Rules

An at-rule is a special CSS rule that acts as a directive controlling the behavior of CSS. It's called an at-rule because it starts with the "at" symbol (`@`). Here are some examples of at-rules:

- `@import`: Includes the contents of another stylesheet.
- `@media`: Defines a media query. Chapter 11 will cover media queries in more detail, but we'll see a few before then.

- `@keyframes`: Defines a set of keyframes for a CSS animation. Chapter 9 will cover animations.
- `@supports`: Conditionally apply CSS rules based on the browser support of a given CSS feature.

How CSS Is Used

There are several ways to use CSS in an HTML document. They all have the same result: a stylesheet that is applied to elements in the document.

Inline Styles

HTML elements support the `style` attribute, where CSS properties can be specified as inline styles. An inline style does not contain selectors or curly braces – it is a collection of CSS properties to be applied to that element only. Listing 1-4 has an example of an inline style.

Listing 1-4. An element with an inline style

```
<div style="background-color: red;">
  Hello world!
</div>
```

The element in Listing 1-4 will have a red background. If there are conflicts in the rules that apply to an element from CSS stylesheets, the inline style always takes precedence. For example, if there was a CSS rule somewhere that made all `div` elements have a blue background, this element's inline style would override that and give it a red background.

Internal Stylesheets

CSS rules can also be specified inside a stylesheet within the HTML document itself. This is done by adding CSS rules inside of a `style` element. Style elements are typically added to the document's `head` element and are full stylesheets with selectors and rules.

Listing 1-5. A style element inside an HTML document

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      div {
        background-color: red;
      }
    </style>
  </head>
  <body>
    <div>Hello world!</div>
  </body>
</html>
```

In the document from Listing 1-5, all `div` elements will have a red background.

External Stylesheets

Lastly, CSS rules can also be listed in a stylesheet file with a `.css` extension. This stylesheet is then referenced in the head of the HTML document using a `link` element. In the following example, all `div` elements in the document will have a red background. The HTML document in Listing 1-6 includes the CSS file from Listing 1-7.

Listing 1-6. An HTML document referencing an external stylesheet

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="/path/to/file.css">
  </head>
  <body>
    <div>Hello world!</div>
  </body>
</html>
```

Listing 1-7. A simple external CSS file

```
div {
  background-color: red;
}
```

Browser Support

The CSS features discussed in this book are well supported in modern browsers: recent versions of Chrome, Firefox, Edge, and Safari. Older browsers, such as Internet Explorer, are not covered. Some features may not be fully supported yet in all browsers at the time of publication.

Some features are supported with vendor-specific prefixes, but this is rare with modern browsers. For example, consider the `backdrop-filter` property. In some WebKit-based browsers, this is only supported with the `-webkit-` prefix, as shown in Listing 1-8.

Listing 1-8. Using vendor prefixes inside a CSS rule

```
.blur-bg {
  /* Standard property */
  backdrop-filter: blur(10px);

  /* Property with a vendor prefix */
  -webkit-backdrop-filter: blur(10px);
}
```

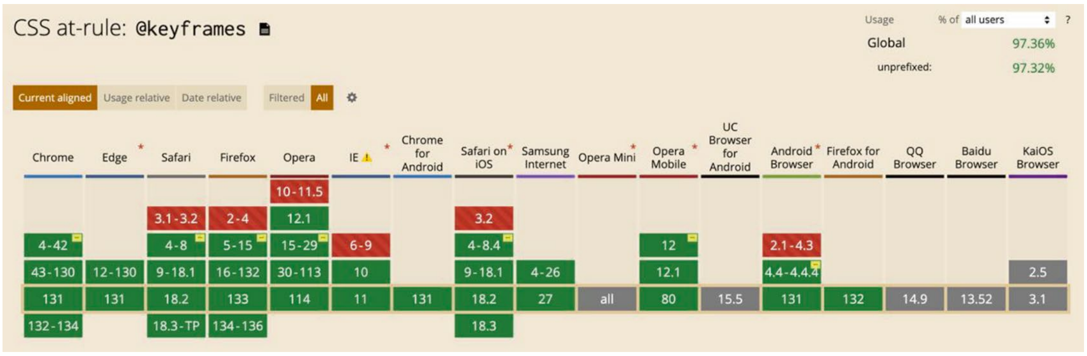
More recent browser versions tend to use experimental feature flags rather than requiring vendor prefixes in the CSS itself. These are special configuration flags that are exposed in an advanced configuration interface, or through special command-line arguments, where experimental features can be turned on or off.

Web Resources

There are many resources and references that are useful when developing CSS. Here are a few of the most popular.

CanIUse.com

The website CanIUse.com (https://caniuse.com) is a great resource for finding out what browsers support a given feature. This site maintains a database of up-to-date browser support information for different CSS features. Figure 1-2 shows a screenshot of an example query.



CSS at-rule: `@keyframes`

Usage % of all users

Global	97.36%
unprefixed:	97.32%

Chrome	Edge	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS	Samsung Internet	Opera Mini	Opera Mobile	UC Browser for Android	Android Browser	Firefox for Android	QQ Browser	Baidu Browser	KaiOS Browser
				10-11.5												
		3.1-3.2	2-4	12.1			3.2									
4-42		4-8	5-15	15-29	6-9		4-8.4			12		2.1-4.3				
43-130	12-130	9-18.1	16-132	30-113	10		9-18.1	4-26		12.1		4.4-4.4.4				2.5
131	131	18.2	133	114	11	131	18.2	27	all	80	15.5	131	132	14.9	13.52	3.1
132-134		18.3-TP	134-136				18.3									

Figure 1-2. *Screenshot of CanIUse.com*

MDN Web Docs

Another useful resource is MDN Web Docs (https://developer.mozilla.org), which has a complete reference to HTML, CSS, JavaScript, and more. It has, among other information, an exhaustive reference of all CSS properties.

How CSS Works in the Browser

Now let's look at how the browser renders a page with CSS.

The Document Object Model (DOM)

The Document Object Model, or DOM, is a data structure in the browser. It is a tree of objects that represent the elements in the document and their structure and hierarchy. This tree is composed of DOM nodes. The DOM is created by reading the HTML markup, tokenizing it, parsing it, and finally creating the object hierarchy.

Consider the HTML document shown in Listing 1-9.

Listing 1-9. A simple HTML document

```
<html>
  <body>
    <h1>Hello World</h1>
    <div>
      <h2>Subtitle</h2>
      <p>Hello world!</p>
    </div>
  </body>
</html>
```

The corresponding DOM tree is shown in Figure 1-3.

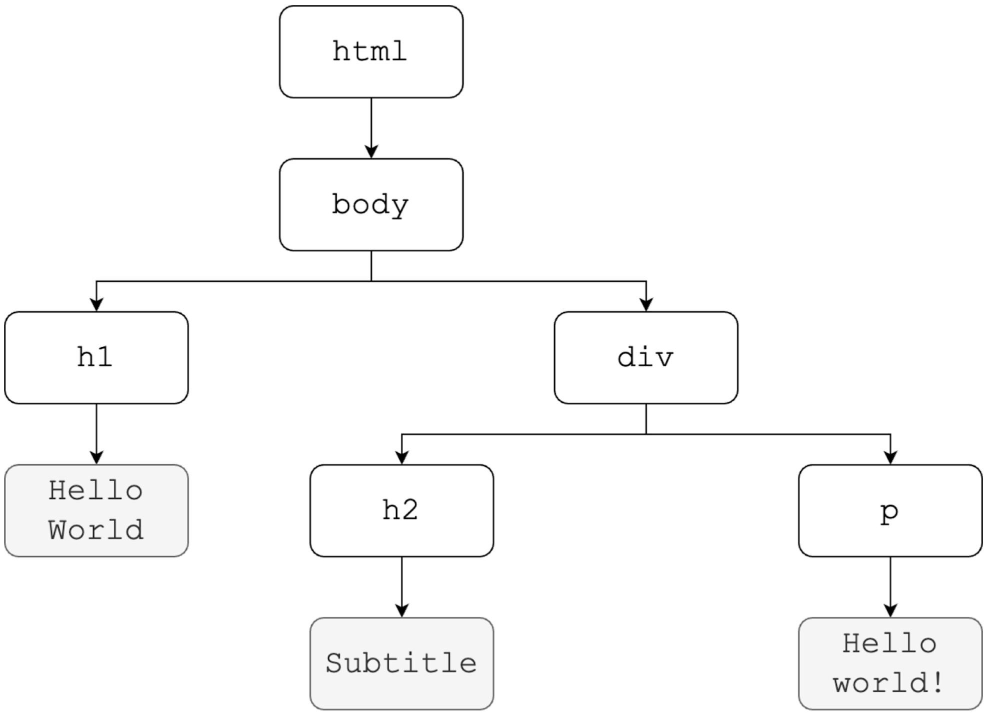


Figure 1-3. *The DOM tree corresponding to the HTML document*

The CSS Object Model (CSSOM)

Like the DOM, there is also a CSS Object Model, or CSSOM. This is another tree structure that represents the hierarchy of styles in the document. While they are both tree structures, the CSSOM is a separate structure from the DOM.

Listing 1-10 contains a CSS stylesheet meant to be applied to the HTML document in Listing 1-9.

Listing 1-10. A CSS stylesheet

```css
body {
  font-size: 16px;
}

h1 {
  font-size: 1.5rem;
  color: orangered;
}

div {
  padding: 1rem;
}

div h2 {
  font-size: 1.2rem;
  color: blue;
}

div p {
  font-size: 0.9rem;
  color: gray;
}
```

The browser parses the CSS, which blocks the rendering of the page, and creates the CSSOM. Figure 1-4 shows the structure of the CSSOM tree.

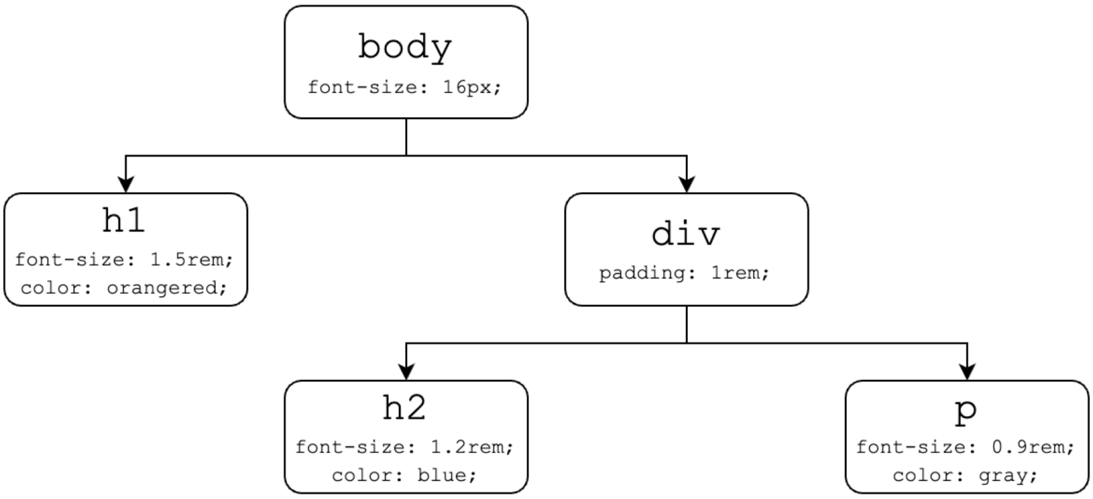


Figure 1-4. *The CSSOM tree*

The Render Tree

Once the DOM and CSSOM are complete, they are combined to form the render tree. The render tree contains all the information the browser needs to render the page. To do this, the browser calculates which CSS rules apply to which elements in the DOM.

Figure 1-5 shows the render tree resulting from combining the DOM and CSSOM trees.

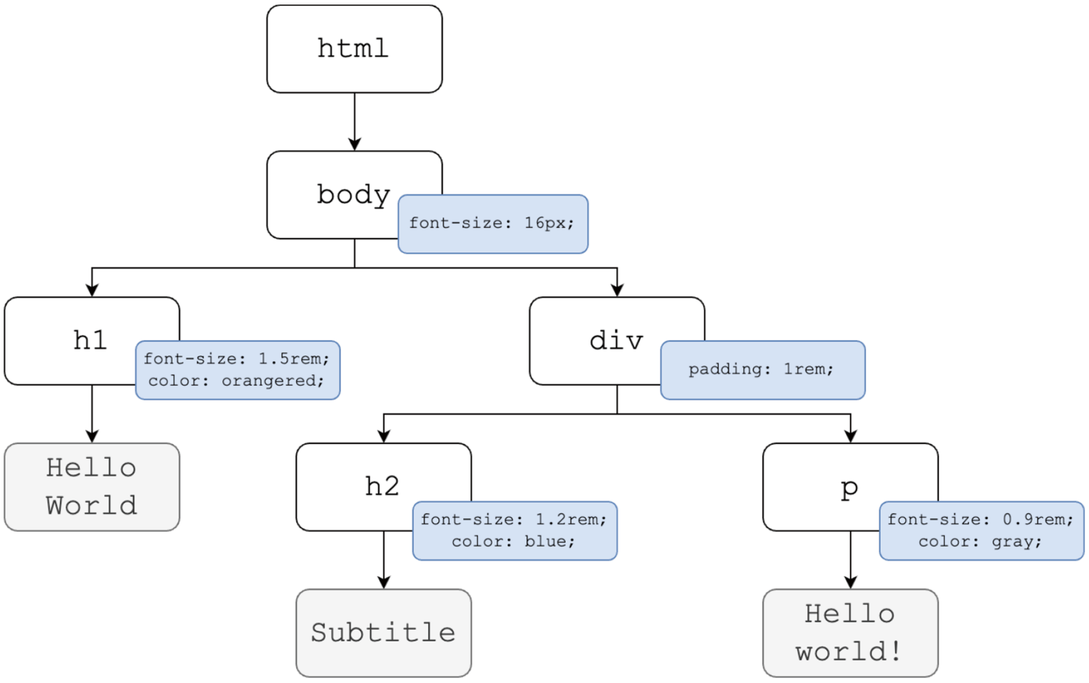

html
body
font-size: 16px;
h1
font-size: 1.5rem; color: orangered;
div
padding: 1rem;
Hello World
h2
font-size: 1.2rem; color: blue;
p
font-size: 0.9rem; color: gray;
Subtitle
Hello world!

Figure 1-5. *The combined render tree*

Layout and Paint

Once the browser has created the render tree, it can begin laying out the elements on the page. This stage of the process looks at styles such as width, height, position, margin, and padding, to determine each element's size and location on the page. At the layout stage, however, nothing is shown on screen yet.

Once layout is complete, the browser can begin painting by applying styles such as color and font to determine the actual pixels to draw on the screen.

Summary

- CSS stylesheets are made up of rules, selectors, properties, and values.
- At-rules are special directives that start with an @ character.
- When there is a conflict, the last style defined wins.

- CSS can be added with inline styles, a `style` element within an HTML document, or an external CSS file.
- Some experimental CSS properties may need vendor prefixes or feature flags.
- The browser builds a tree structure of the DOM and CSSOM, then combines them to form the render tree.

CHAPTER 2

CSS Rules and Selectors

A *selector* determines which element(s) a CSS rule applies to. There are several ways an element can be targeted with a selector, which we will cover in this chapter.

CSS selectors can match multiple elements on the page. That is, a single CSS rule can apply to multiple elements. An element or class selector can select multiple elements that have that element or class name.

Similarly, a single HTML element can be affected by multiple CSS rules. An element will have the properties from all applicable CSS rules applied to it. Sometimes, multiple CSS rules might have conflicting properties. This is where the concept of specificity comes in, which we will also cover in this chapter.

Basic Selector Types

The basic types of selectors are

- Universal
- Element
- ID
- Class
- Attribute

The Universal Selector

The universal selector, specified simply as an asterisk (*), matches all elements. This can be specified as a single selector, to select all elements in the document, or with combinators (discussed below). Listing 2-1 shows an example usage of the universal selector.

J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_2

Listing 2-1. Removing all margins with the universal selector

```
* {
  margin: 0;
}
```

The CSS rule in Listing 2-1 will apply a `margin` of `0` to all elements in the document.

Element Selectors

An element selector targets an HTML element by its element, or tag, name. The syntax of the selector is the name of the element. Listing 2-2 shows an example of an element selector.

Listing 2-2. Applying a `margin` to all `p` elements

```
p {
  margin: 25px;
}
```

The CSS rule in Listing 2-2 will apply a margin of `25px` to all `p` elements in the document.

ID Selectors

An HTML element can have an `id` attribute. As a rule, there should only be one element with a given `id`. If there are multiple elements with the same `id`, most browsers will match the rule with all elements having that `id`. However, this should be avoided as it violates the HTML specification.

An ID selector is specified with the # character followed by the `id` value, as shown in Listing 2-3.

Listing 2-3. Applying `padding` to the element with an `id` of `header`

```
#header {
  padding: 25px;
}
```

The element with an `id` attribute whose value is `header` receives `25px` of padding. If there are other elements also having an `id` of `header`, they also receive `25px` of padding (in most browsers). Again, this should be avoided. If you need to apply a style to more than one element, you can use class selectors instead of ID selectors.

Class Selectors

An HTML element can also have a `class` attribute. A class can be used to mark all elements of a related type.

While only a single element is intended to be targeted by an ID selector, any number of HTML elements can have the same `class` attribute. Similarly, a single HTML element can have any number of classes applied to it. Multiple classes are separated by a space in the value of the element's `class` attribute.

A class selector will match every element in the document with the given class. Class selectors are specified with a dot, followed by the name of the class, as shown in Listing 2-4.

Listing 2-4. Applying a color to all elements with the class `nav-link`

```
.nav-link {
  color: darkcyan;
}
```

The rule in Listing 2-4 will match every element in the document with a class of `nav-link` and give it a color of `darkcyan`.

Attribute Selectors

HTML elements can also be selected by their attribute values or by the presence of an attribute. The attribute is specified inside square brackets, and the attribute selector can take several forms:

`[name]`
Selects all elements that have the given attribute, regardless of its value.

`[name="value"]`
Selects all elements that have the given attribute, whose value is the exact string `value`.

`[name~="value"]`
Selects all elements that have the given attribute, whose value contains the string `value` separated by whitespace. Listing 2-5 contains two HTML elements with slightly different `title` attributes.

Listing 2-5. Example HTML elements

```
<div title="Hello World">Hello World</div>
<div title="HelloWorld">HelloWorld</div>
```

If you wrote a CSS rule with the selector `[title~="World"]`, the first element would match but not the second. This is because in the second element, the word "World" in the `title` attribute is not surrounded by whitespace.

`[name*="value"]`
Selects all elements that have the given attribute, whose value contains the substring `value`. If we wrote another CSS rule, this time with the selector `[title*="World"]`, it would match both elements from Listing 2-5.

`[name^="value"]`
Selects all elements that have the given attribute, whose value begins with `value`.

`[name$="value"]`
Selects all elements that have the given attribute, whose value ends with `value`.

Compound Selectors

Any of the above selectors (except for the universal selector) can be used alone or in combination with other selectors to make the selector more specific. This is best illustrated with some examples:

`div.my-class`
Matches all `div` elements with a class of `my-class`.

`span.class-one.class-two`
Matches all `span` elements with a class of *both* `class-one` and `class-two`.

`a.nav-link[href*="example.com"]`
Matches all `a` elements with a class of `nav-link` that have an `href` attribute containing the string `example.com`.

Multiple Independent Selectors

A CSS rule can have multiple selectors separated by a comma. The rule will be applied to any element that is matched by any one of the given selectors:

`.class-one, .class-two`
Matches all elements with a class of `class-one` as well as all elements with a class of `class-two`.

Selector Combinators

Combinators are used to make more specific selectors. Combinators work alongside the basic selectors discussed previously. For a given rule, multiple basic selectors can be used, joined by a combinator.

Descendant Combinator

The descendant combinator matches an element that is a descendant of the element on the left-hand side. Descendant means that the element exists somewhere within the child hierarchy – it does not have to be a direct child.

The descendant combinator is specified with a space character, as shown in Listing 2-6.

Listing 2-6. An example of the descendant combinator

```
.header div {
  color: green;
}
```

This selector matches all `div` elements that are direct or indirect descendants of any element with the class `header`. If these `div` elements contain additional `div` elements as children, those child `div` elements will also be matched by the selector.

Child Combinator

The child combinator matches an element that is a *direct child* of the element on the left-hand side. It is specified with a > character, as shown in Listing 2-7.

Listing 2-7. An example of the child combinator

```
.header > div {
  color: green;
}
```

This selector matches all `div` elements that are direct children of an element with the class `header`. If those `div` elements contain additional `div` elements as children, those `div` elements will *not* be matched by the selector (because they are descendants, not direct children).

Subsequent Sibling Combinator

The subsequent sibling combinator matches an element that is a sibling, but not necessarily an *immediate* sibling, of the element on the left-hand side. It is specified with a ~ character. It only matches siblings going forward and does not match previous siblings. Consider the example in Listing 2-8.

Listing 2-8. Example HTML elements

```
<div>
  <div class="intro"></div>
  <div class="article1"></div>
  <div class="article2"></div>
  <div class="summary"></div>
</div>
```

Listing 2-9 has an example of using the subsequent sibling combinator.

Listing 2-9. The subsequent sibling combinator

```
.article1 ~ div {
  background: skyblue;
}
```

When applied to the HTML in Listing 2-8, this selector would match the `div` elements with the classes `article2` and `summary` because these are subsequent siblings. It would not match the previous sibling, the div with the class `intro`.

Next Sibling Combinator

The next sibling combinator, specified with a + character, matches an element's immediate next sibling only.

Listing 2-10 has an example of applying the next sibling combinator to the HTML in Listing 2-8.

Listing 2-10. The next sibling combinator

```
.article1 + div {
  background: skyblue;
}
```

When applied to the HTML in Listing 2-8, only the `article2` element would be matched because it is the next immediate sibling of the `article1` element.

Using Multiple Combinators

Just like basic selectors, combinators can be used together to form even more specific selectors. For example, consider the HTML from Listing 2-11.

Listing 2-11. Example HTML for multiple combinators

```
<div>
  <header>
    <div>Title</div>
    <button>Button</button>
  </header>
</div>
```

Listing 2-12 uses multiple combinators to form a more complex selector.

Listing 2-12. A selector with multiple combinators

```
header > div + button {
  background: gray;
}
```

This selector matches the `button` element inside the `header`. This works by first selecting the `header`'s immediate child, the `div`, then selecting that element's next sibling, the `button`.

Pseudo-classes

A pseudo-class allows you to select elements based on some special state of the element, in addition to all the selectors discussed previously. Pseudo-classes start with a colon (`:`) character and can be used alone or combined with other selectors.

Some pseudo-classes let you select elements based on interaction state (such as `:focus` or `:hover`), while others let you select elements based on their position in the document (such as `:first-child`).

Pseudo-classes can be used on their own (e.g., `:hover`) or can be *anchored* to another element (e.g., `button:hover`, `.my-button:hover`).

There are many pseudo-classes (you can find a complete list at `https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes`), but here are some commonly used ones.

Interaction State

These pseudo-classes are based on some user interaction state:

`:active`
Matches an element that is currently being activated. For buttons and links, this usually means the mouse button has been pressed, but not yet released.

`:checked`
Matches a radio button, checkbox, or option inside a `select` element that is checked or selected.

:focus
Matches an element that currently has the input focus. This is typically used for buttons, links, and text fields.

:focus-within
Similar to `:focus` but also matches an element that has a descendant element that is currently focused.

:hover
Matches an element that the mouse cursor is currently hovering over. This is typically used for buttons and links but can be applied to any type of element.

:valid, :invalid
Used with form elements using the HTML validation API. `:valid` matches an element which is currently valid according to validation rules, and `:invalid` matches an element which is not currently valid.

:visited
Matches a link whose URL has already been visited by the user. To protect a user's privacy, the browser limits what styling can be applied to an element matched by this pseudo-class.

Document Structure

These pseudo-classes are based on an element's position in the document:

:first-child, :last-child
Matches an element that is the first or last child of its parent. Consider the example unordered list in Listing 2-13.

Listing 2-13. A simple unordered list

```
<ul class="my-list">
  <li>Item one</li>
  <li>Item two</li>
</ul>
```

The selector `.my-list > li:first-child` will match the first list item only, and the selector `.my-list > li:last-child` will match the last list item only.

`:nth-child(n)`

This pseudo-class takes an argument. It matches an element that is the *n*th child of its parent. The index of the children starts at 1. Referring to Listing 2-13, you could also select the first item with the selector `.my-list > li:nth-child(1)` or the second item with the selector `.my-list > li:nth-child(2)`.

The `:nth-child` pseudo-class can also select children at a given interval. For example, in a longer list, you could select every other list item with the selector `.my-list > li:nth-child(2n)`. Or you could select every fourth item with the selector `.my-list > li:nth-child(4n)`.

You can even select even or odd numbered children with the pseudo-classes `:nth-child(even)` or `:nth-child(odd)`.

`:nth-of-type(n)`

Similar to `:nth-child`, except that it only considers children of the same type. For example, the selector `div:nth-of-type(2)` matches any `div` element that is the second `div` element among any group of children.

`:root`

Matches the root element of the document. This is usually the `html` element. This selector can be useful for several reasons, one of which is that it can be used to declare global variables (we will discuss CSS variables in Chapter 3).

Negating a Selector

A selector can also include the `:not()` pseudo-class. `:not` accepts a selector as its argument and will match any element for which the selector does not match. For example, the selector `div:not(.fancy)` will match any `div` that does *not* have the `fancy` class.

The :has Pseudo-class

Up to now, we've discussed selectors that let you select subsequent/next sibling or descendant/child elements, but until recently there wasn't a way to select a parent or previous sibling element. With the `:has` pseudo-class, this is now possible.

`:has` takes an argument which is one or more *relative selectors*, which typically start with a combinator. The selector matches if the relative selector matches, starting at the element anchored to the pseudo-class. For an example, consider the HTML shown in Listing 2-14.

Listing 2-14. Some example HTML markup

```
<div class="card">
  Card 1
</div>

<div class="card">
  <img src="https://placehold.co/400">
  Card 2
</div>
```

Suppose you want to give each card a border, and you want cards that contain an image to have some extra padding. You can use the `:has` pseudo-class to accomplish this, as shown in Listing 2-15.

Listing 2-15. CSS using the `:has` pseudo-class

```
.card {
  border: 1px solid gray;
  padding: 4px;
}

.card:has(> img) {
  padding: 8px;
}
```

Both cards will get a gray border. Card 1 gets `4px` of padding due to the first CSS rule. The `:has` pseudo-class on the second rule's selector matches the card that contains the image as its direct child and applies `8px` of padding.

You can also use `:has` to select a previous sibling element. Listing 2-16 shows part of a form. It has a required `input` field preceded by a `label` element.

Listing 2-16. Required form field with a label

```
<div>
  <label for="name">Name:</label>
  <input id="name" type="text" required />
</div>
```

Suppose you want to apply a red border to the field when it's invalid, and you also want to make the label appear red. You can use `:has` to select the `label` preceding the invalid `input` field, as shown in Listing 2-17.

Listing 2-17. Form validation CSS using the `:has` pseudo-class

```
:invalid {
  border: 1px solid red;
}

label:has(+ :invalid) {
  color: red;
}
```

The first selector gives the invalid `input` field a red border. The second selector finds the `label` whose next sibling is the invalid `input` field and makes the color red. Once the `input`'s value becomes valid, this rule no longer applies, and the red color is removed.

The :is Pseudo-class

The `:is` pseudo-class takes a comma-separated list of selectors as its argument and matches any element that matches one of the specified selectors. This can make a rule that has multiple nested selectors a little more concise. Listing 2-18 shows an example of simplifying multiple nested selectors with the `:is` pseudo-class.

Listing 2-18. The `:is` pseudo-class

```
/* Without the :is pseudo-class */
nav ul li a, nav ol li a {
  text-decoration: none;
}
```

```
/* Using the :is pseudo-class */
nav :is(ul, ol) li a {
  text-decoration: none;
}
```

Pseudo-elements

A pseudo-element lets you select parts of the HTML document that don't have a corresponding DOM element to select. Sometimes, this can be part of a matched element, and other times it can be separate content without a DOM element. Pseudo-elements are specified with a double colon (`::`) followed by the pseudo-element name.

We haven't discussed the differences between block and inline elements yet, but when we do, keep in mind that some pseudo-elements only apply to block elements.

`::first-line`
Matches the first line of a block element.

`::first-letter`
Matches just the first letter of the first line of a block element.

`::before, ::after`
Two special pseudo-elements are `::before` and `::after`. These pseudo-elements don't select part of the element; rather, they create a new element as either the first child or the last child of the matched element, respectively. These pseudo-elements are typically used to decorate or add effects to an element.

Suppose you want to add an indicator next to all external links on your website. You can tag these external links using a class, such as `external-link`. You can specify an external link as shown in Listing 2-19.

Listing 2-19. An external link

```
<a class="external-link"
   href="https://google.com">
  Google
</a>
```

Then you can add the indicator with the CSS rule in Listing 2-20. The `content` property defines what the text content of the pseudo-element should be.

Listing 2-20. Adding the external link indicator

```
.external-link::after {
  content: ' (external)';
  color: green;
}
```

Figure 2-1 shows the rendered HTML.

Figure 2-1. *The rendered link with an ::after pseudo-element*

The `::after` pseudo-element added the content `(external)` and made it green. Sometimes, you may want to use a `::before` or `::after` pseudo-element for decorative purposes. In this case, you must still provide a value for the `content` property, or else the element will not be displayed. For decorative elements, this can be set to an empty string.

Specificity

An HTML element can have multiple CSS rules applied to it by matching different selectors. What happens if two or more of the rules applied to an element contain the same CSS property? How are such conflicts resolved? Consider the HTML in Listing 2-21, a simple `div` with a class.

Listing 2-21. Sample HTML markup

```
<div class="profile">My Profile</div>
```

Now, suppose you have the CSS in Listing 2-22 applied to this HTML.

Listing 2-22. Conflicting CSS rules

```
.profile {
  background-color: green;
}

div {
  background-color: red;
  color: white;
}
```

We have a conflict. The HTML element matches both selectors – it is indeed a `div` element, and it also has the `profile` class. Each rule specifies a different value for the `background-color` property. When the page is rendered, which background color will this `div` element have? Figure 2-2 shows the output of this code.

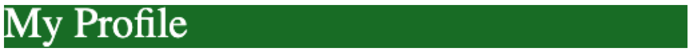


Figure 2-2. *The rendered output of the conflicting CSS*

The class selector rule's background color was applied. This is because the class selector rule has a higher *specificity*. When there is a conflict of CSS properties across multiple rules, the rule with the most specific selector will be chosen. According to the rules of CSS, a class selector is more specific than an element selector.

Note that while the element has the background color from the class selector rule, it also has the color from the element selector rule. Specificity rules only matter for conflicting properties across multiple rules. Other properties in these multiple rules will still be applied.

Specificity Rankings

The specificity rankings of CSS rules are as follows, from most specific to least specific:

1. ID selectors
2. Class selectors, attribute selectors, and pseudo-classes
3. Element selectors and pseudo-elements

Neither the universal selector nor combinators factor into specificity. You can make a rule more specific by combining multiple rules with a combinator, but the combinator itself does not have a value when specificity is calculated.

Calculating Specificity

There is a general algorithm for calculating a CSS rule's specificity based on the types of selectors it contains. Imagine three boxes, one for each type of style rule in the list above. Initially, each box has a zero in it, as shown in Figure 2-3.

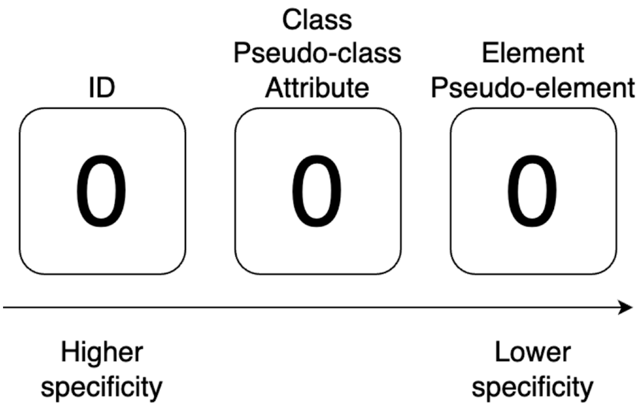


Figure 2-3. *Specificity calculator*

For each ID in the selector, add 1 to the value in the first box. For each class, pseudo-class, or attribute in the selector, add 1 to the second box. Finally, for each element or pseudo-element in the selector, add 1 to the value in the last box.

To compare two selectors and find the higher specificity, compare the values of the three boxes between each selector. The calculation works by finding the first box, moving from left to right, with differing values. Once you find differing values, the selector with the higher value for that box has a higher specificity.

If all boxes have the same value, then the selectors are considered to have equal specificity. This is best illustrated with an example. Consider the following two selectors:

- `#menu .item`
- `.navbar .link`

To find which has the higher specificity, let’s break them down into the three boxes as shown in Figure 2-3. This breakdown is shown in Figure 2-4.

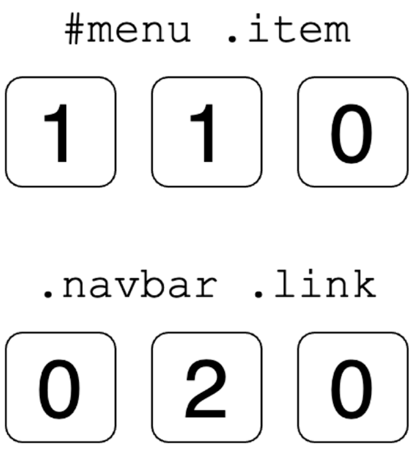


Figure 2-4. *Breaking down the selectors to compare specificity*

The selector `#menu .item` has one ID selector and one class selector, so its values are 1-1-0. The other selector, `.navbar .link`, only has two class selectors, so its values are 0-2-0. Here, the first selector wins because it has a higher value for the first box.

If an element has inline styles in a `style` attribute, the inline styles will take precedence over properties from a CSS rule, regardless of its specificity, unless a property has the `!important` keyword after it.

The Escape Hatch: `!important`

Any CSS property can have the keyword `!important` after it inside of a rule. This keyword will cause that property to always win in a conflict, even if the rule that contains it has lower specificity than another conflicting rule. An example of this is shown in Listing 2-23.

Listing 2-23. Using the !important keyword

```
#menu .item {
  color: red;
}

.navbar .link {
  color: blue !important;
}
```

Even though the first selector is more specific, if both selectors match the same element, the color will be blue. It will win the conflict because of the `!important`. `!important` will even override an element's inline styles.

This is generally considered a bad practice. It can make CSS issues harder to debug and can make your style sheets less maintainable. In most cases, it's better to determine the specificity of the rules you are trying to apply and use a more specific selector on the rule that you want to apply.

Nesting CSS Rules

CSS rules can be nested inside other rules. In the past, this was possible only by using a preprocessor such as Sass or LESS, but modern browsers now support nesting rules. Nesting rules can improve readability and reduce repetitive selectors. For an example of this, see Listing 2-24, which shows the HTML for a card component.

Listing 2-24. A simple card component

```
<div class="card">
  <h2 class="card-title">My Card</h2>
  <p>This is my card's content.</p>
  <button class="card-button">Learn More</button>
</div>
```

Listing 2-25 has some CSS rules to apply to this card component, first without using nested rules.

Listing 2-25. Styling the card component

```
.card {
  background: #cccccc;
}

.card .card-title {
  color: red;
}
```

```
.card .card-button {
  background: #aaaaaa;
}

.card .card-button:hover {
  background: #eeeeee;
}
```

The CSS in Listing 2-25 adds some background and text color styling and changes the background color of the button on hover. Let's take these CSS rules and convert them to an equivalent set of nested rules, shown in Listing 2-26.

Listing 2-26. Styling the card with nested rules

```
.card {
  background: #cccccc;

  .card-title {
    color: red;
  }

  .card-button {
    background: #aaaaaa;

    &:hover {
      background: #eeeeee;
    }
  }
}
```

When a rule is nested inside another rule, that selector is treated as a child of the parent selector. For example, the `card-title` class selector will match an element with this class that is a child of `card`. Notice the `&` symbol before the `:hover` pseudo-class, though. This is the *nesting selector* and is used to join the `:hover` pseudo-class to the `.card-button` class, instead of selecting a child.

Summary

- The most used selector types are element, ID, class, and attribute selectors.
- Combinators can be used to create more specific selectors.
- Multiple selectors can match the same element.
- Pseudo-classes and pseudo-elements give you finer control over what elements are selected.
- Conflicts are resolved by using the rule with a higher specificity.
- For better readability, CSS rules can be nested inside other rules.
- Specificity can be overridden by using `!important`, but this is not recommended.

CHAPTER 3

Basic CSS Concepts

Now that we've looked in detail at how to select elements, let's start to explore how to style them. The next step is to look at some of the basic concepts in CSS.

The Box Model

Every element in CSS is treated like a rectangular box. This is known as the *box model*. The box is made up of four parts. Starting from the outside and moving toward the center, these are

- *Margin*: Outer spacing between this element's border and its surrounding elements. It's specified with the `margin` property.
- *Border*: An outline around the element. Borders can have a thickness, style, and color. It's specified with several properties: `border-style`, `border-width`, `border-color`, and the shorthand `border` property.
- *Padding*: Inner spacing between the border and the content, specified with the `padding` property.
- *Content*: Main area where the element's content goes. This is controlled with the `width` and `height` properties.

Figure 3-1 shows the different parts of the CSS box model.

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_3

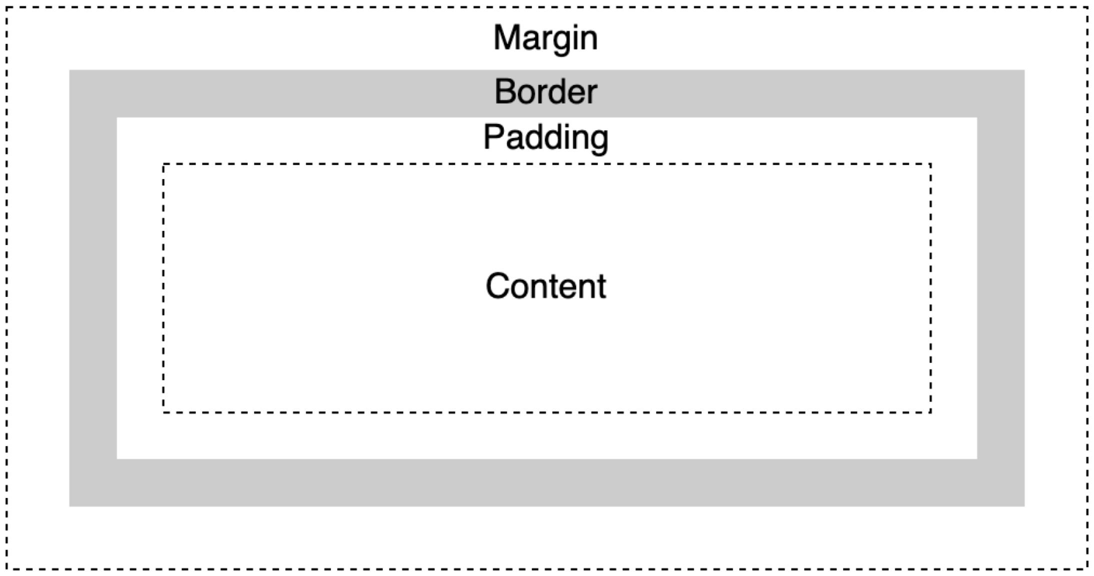


***Figure 3-1.** The CSS box model*

By default, most elements have no padding, border, or margin. However, there are some exceptions, like `button` or `ul` elements. Browsers have a built-in stylesheet that apply padding, margin, and other default styles to certain HTML elements.

Listing 3-1 shows two `div` elements with some color to differentiate them, but no other styling is applied.

Listing 3-1. Two simple div elements

```
<style>
  #div1 {
    background-color: #fca5a5;
  }

  #div2 {
    background-color: #86efac;
  }
</style>

<div id="div1">Hello world!</div>
<div id="div2">Hello world!</div>
```

Figure 3-2 shows the result of rendering these two elements.

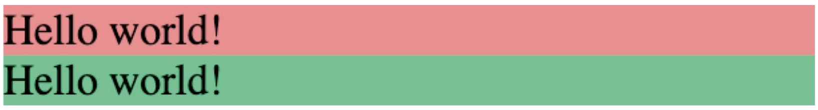


Figure 3-2. *The rendered result*

Notice how the elements run up against one another and generally look cramped. By adding padding, border, and margin to these elements, you can make it easier to read. Listing 3-2 adds these properties.

Listing 3-2. Applying padding, border, and margin

```
<style>
  #div1 {
    background-color: #fca5a5;
  }

  #div2 {
    background-color: #86efac;
  }

  #div1, #div2 {
    padding: 1rem;
    margin: 1rem;
    border: 1px solid #000000;
  }
</style>

<div id="div1">Hello world!</div>
<div id="div2">Hello world!</div>
```

Figure 3-3 shows the elements with their new styling.

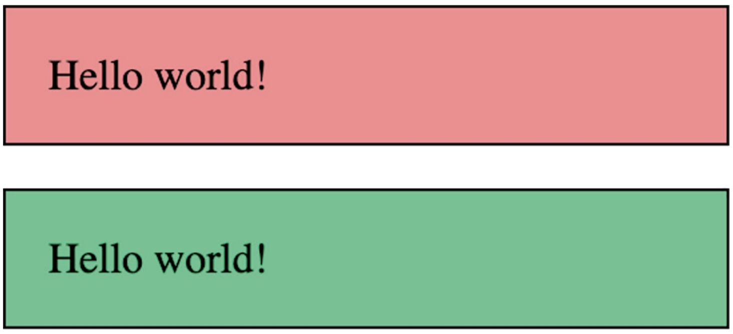


Figure 3-3. *The same two elements with padding, border, and margin applied*

The elements now have inner and outer spacing, giving them improved readability. They are not as close together with the new styles.

Box Sizing

An element's size is specified with the `width` and `height` properties. How exactly this is interpreted, however, depends on the value of the `box-sizing` property. This property supports two values: `content-box` and `border-box`.

Box Sizing with `content-box`

This is the default. With `content-box`, the `width` and `height` properties are treated as the width and height of the *content area* of the box only. The actual dimensions of the element's box are the sum of its width and height (the content box), the padding on each side, and the border width on each side.

For an example of this, see the rule in Listing 3-3.

Listing 3-3. A simple CSS rule using `box-sizing: content-box`

```css
.box {
  width: 100px;
  height: 100px;
  padding: 10px;
  border: 5px solid red;
  box-sizing: content-box;
}
```

The content area of the box has a width and height of 100 pixels. The padding (10 pixels on each side) and the border (5 pixels on each side) are added to the element's size, resulting in a total size of 130 pixels (`100px + 10px + 10px + 5px + 5px = 130px`). This is visualized in Figure 3-4.

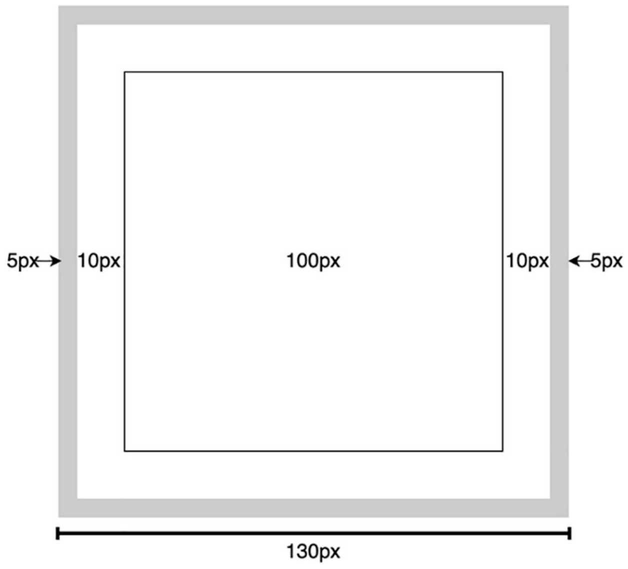


Figure 3-4. *The full rendered size of the element using* `box-sizing: content-box`

Box Sizing with `border-box`

With `border-box`, the values of the `width` and `height` properties are treated as the size of the content box *plus* the padding and border width. Consider the example box in Listing 3-4, which uses `border-box`.

Listing 3-4. A box using `box-sizing: border-box`

```
.box {
  width: 100px;
  height: 100px;
  padding: 10px;
  border: 5px solid red;
  box-sizing: border-box;
}
```

This box has a total width and height of 100 pixels. To compensate for the extra 30 pixels taken up by the padding and border, the content box shrinks to 70 pixels, as visualized in Figure 3-5.

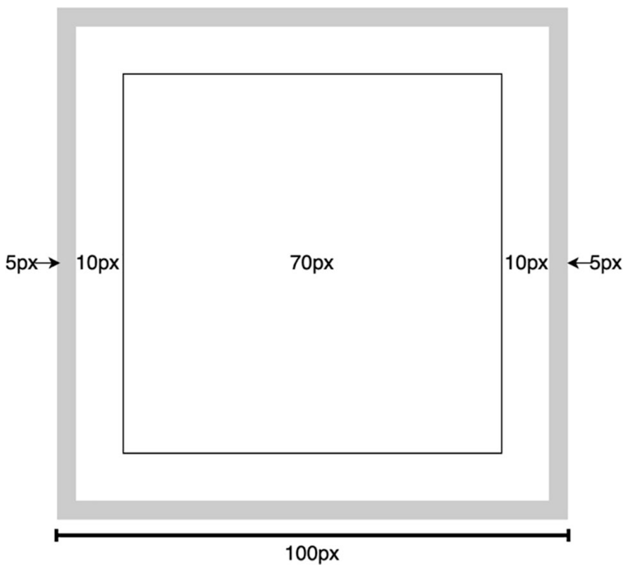


Figure 3-5. *The full rendered size of the element using* `box-sizing: border-box`

Aspect Ratio

Sometimes, you might not want a specific width and height for an element, but rather a particular ratio between the width and height. This is known as the *aspect ratio* and is controlled with the `aspect-ratio` property.

For example, suppose you want an element to always have an aspect ratio of 16:9 (a common aspect ratio for widescreen devices). To accomplish this, you can specify `aspect-ratio: 16 / 9`. The element's width, height, or both must be set to `auto` to let the browser automatically calculate the proper dimensions to meet the needed aspect ratio. If you don't specify `auto` for either of these properties, the `aspect-ratio` is ignored.

Figure 3-6 shows examples of element sizes using three common `aspect-ratio` values with a height of `200px`.

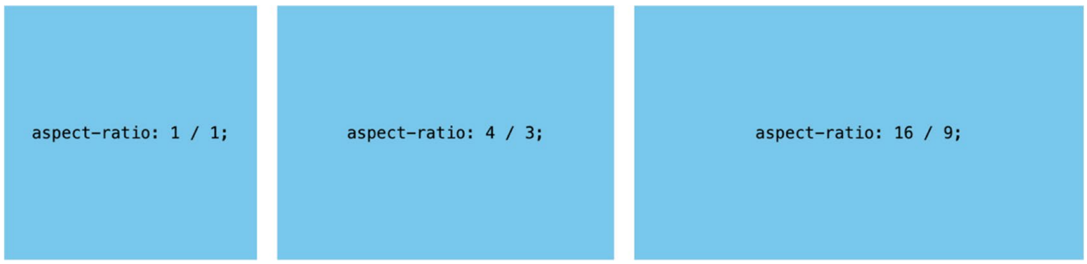


Figure 3-6. *Aspect ratio examples of 1:1, 4:3, and 16:9*

Block and Inline Elements

There are two types of HTML elements: `block` and `inline` (there's also a third type, `inline-block`, which combines aspects of both). Both block and inline elements follow the box model but are different in some important ways.

Some HTML elements default to being block elements, such as a `div` element. Others default to inline, such as a `span` element. These are just defaults; any element can be changed by setting its `display` property to `block`, `inline`, or `inline-block`. The following sections explain the differences between block and inline elements.

Block Elements

Block elements are laid out vertically and take up the full width of their containing element, unless an explicit width is set with the `width` property. A block element's height, by default, is just enough to fit its content. This height can also explicitly be set with the `height` property.

The example in Listing 3-5 contains some `div` elements, which are block elements.

Listing 3-5. Some styled div elements

```
<style>
  .container {
    width: 350px;
  }

  .box1 {
    background-color: skyblue;
  }
  .box2 {
    background-color: lime;
  }
</style>

<div class="container">
  <div class="box1">Hello</div>
  <div class="box2">World</div>
</div>
```

The rendered result is shown in Figure 3-7.

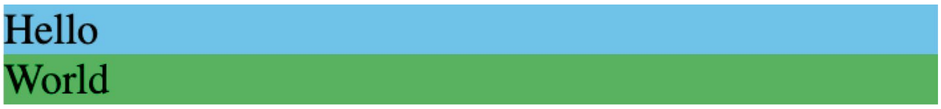


Figure 3-7. *The `div` elements rendered as block elements*

The outer container element has an explicit width of `350px` set, so it is 350 pixels wide. The inner elements have no explicit widths set, so they take up the full width of the container element. They also have no explicit height, so they only take up enough vertical space as needed to fit the text content.

Inline Elements

Unlike block elements, inline elements are rendered inside the normal flow of text. They only take up enough width and height as necessary to contain their content. Setting the `width` or `height` properties of an inline element has no effect. Listing 3-6 has an example of an inline element.

Listing 3-6. An inline span element

```
<style>
  .greeting {
    background: skyblue;
  }
</style>

<span class="greeting">Hello world!</span> I am demonstrating an inline
element.
```

Figure 3-8 shows the rendered result.

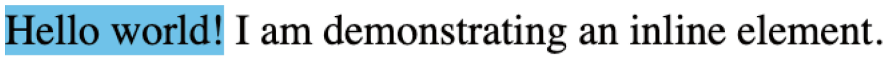


Figure 3-8. *The styled inline element*

The span is an inline element, so its width and height are only enough to fit its content. It does not appear on its own line. If you try to set a different width, as is shown in Listing 3-7, it has no effect.

Listing 3-7. Attempting to set a width on an inline element

```
<style>
  .greeting {
    background: skyblue;
    width: 500px;
  }
</style>

<span class="greeting">Hello world!</span> I am demonstrating an inline
element.
```

As Figure 3-9 shows, the width is unchanged.

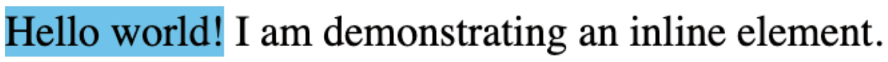


Figure 3-9. *The rendered result, showing no change in the element's width*

Padding and Margin with Inline Elements

There are some other differences, with respect to padding and margin, between block and inline elements. If you add horizontal padding to an inline element, it behaves as expected. Listing 3-8 has an example that illustrates this.

Listing 3-8. Horizontal padding on an inline element

```
<style>
  .greeting {
    background-color: skyblue;
    padding-left: 50px;
    padding-right: 50px;
  }
</style>

<span class="greeting">Hello world!</span> I am demonstrating an inline
element.
```

The result is shown in Figure 3-10.

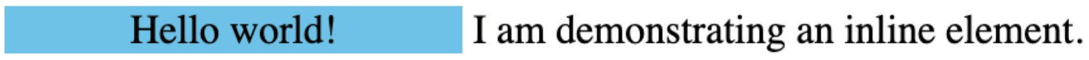


Figure 3-10. *Adding horizontal padding to an inline element*

As expected, the element's width increases to accommodate the padding, and the surrounding content is pushed away to make room. However, an inline element behaves differently when setting vertical padding. Listing 3-9 has an inline element with vertical padding. It's surrounded by a narrow container element so that the text content wraps, to better illustrate what happens with the vertical padding.

Listing 3-9. An inline element with vertical padding

```
<style>
  .container {
    width: 250px;
  }

  .greeting {
    background: skyblue;
    padding-top: 50px;
    padding-bottom: 50px;
  }
</style>

<div class="container">
  <span class="greeting">Hello world!</span> I am demonstrating an inline
  element.
</div>
```

The result, shown in Figure 3-11, may surprise you.

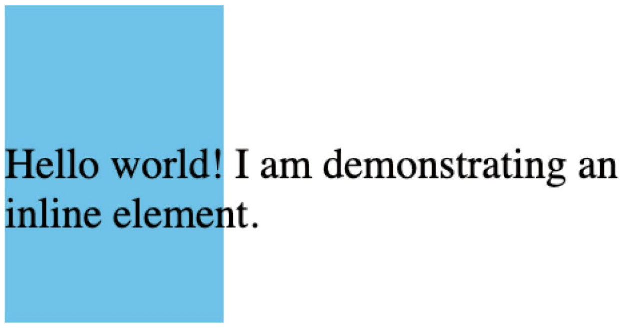


Figure 3-11. *The odd behavior of vertical padding on an inline element*

The vertical padding was applied to the element, but no extra vertical space was made to accommodate this padding. The background color of the greeting element bleeds into the adjacent content.

Inline elements behave similarly when it comes to margins. The horizontal margins are applied, but the vertical margins are not. Listing 3-10 applies horizontal and vertical margin to an inline element, and Figure 3-12 shows the result.

Listing 3-10. Applying margin to an inline element

```
<style>
  .container {
    width: 250px;
  }

  .greeting {
    background: skyblue;
    margin: 50px;
  }
</style>

<div class="container">
  <span class="greeting">Hello world!</span> I am demonstrating an inline
  element.
</div>
```

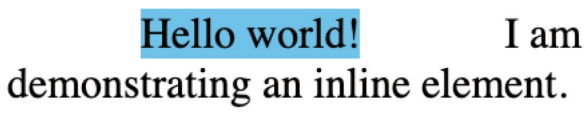


Figure 3-12. *The inline element with margin applied*

Space is added to the left and right of the element for the horizontal margin, but the vertical margin has no effect.

Inline-Block Elements

An inline-block element is a combination of inline and block. The element flows with the text like an inline element, but the `width` and `height` properties are respected, as are vertical padding and margin. Listing 3-11 demonstrates an inline-block element with width, height, padding, and margin.

Listing 3-11. An inline-block element

```
<style>
  .container {
    width: 350px;
    margin-top: 200px;
  }

  .greeting {
    display: inline-block;
    background: skyblue;
    margin: 50px;
    padding: 50px;
    width: 100px;
    height: 100px;
  }
</style>

<div class="container">
  <div>Greeting</div>
  <span class="greeting">Hello world!</span> I am demonstrating an inline
  element.
</div>
```

The result is shown in Figure 3-13.

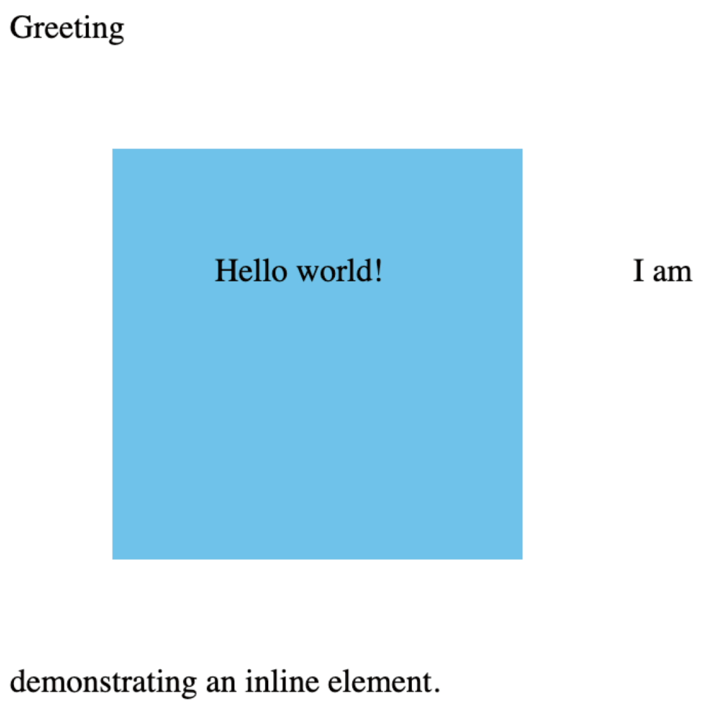


Figure 3-13. *The rendered result of the inline-block element*

The element is rendered with the flow of text like an inline element, but it has proper width, height, padding, and margin like a block element. Extra space is made to account for the padding instead of the element bleeding into the surrounding content like an inline element would.

CSS Units

Many CSS properties represent a size or distance. Just as there are different measurement units in the physical world (such as feet or meters), CSS has different units to specify these sizes and distances (such as `px` or `rem`).

The px Unit

You've already seen the px unit in several examples. In the past, you could have said that px corresponds to physical pixels on the screen. However, in the modern age of ultra-high-resolution displays, this is no longer exactly accurate. A CSS pixel doesn't necessarily have a one-to-one correspondence to a physical device pixel. On a very high-resolution 4K display, a physical pixel would be so tiny that it would be hard to see with the naked eye. If CSS used physical device pixels, then a 1px border would barely be visible.

Instead, a so-called logical pixel corresponds to a certain number of physical device pixels. A 1px border looks roughly equivalent on a 4K display as it does on a lower-resolution display, but it uses more physical device pixels.

The em Unit

The em unit is a relative unit. It is relative to the element's font size. Listing 3-12 shows a rule using the em unit for padding.

Listing 3-12. Using the em unit

```
.header {
  font-size: 24px;
  padding: 0.5em;
}
```

The header element has a font size of 24px. The padding is specified as 0.5em, or half of the element's font size. Therefore, this element has 12px of padding applied, half the font size of 24px.

Listing 3-13 shows another example of using the em unit.

Listing 3-13. Using the em unit with nested elements

```
.header {
  font-size: 24px;
  padding: 0.5em;
}
```

```
.header li {
  font-size: 0.75em;
}

.header li a {
  font-size: 0.5em;
}
```

Because font size is inherited, the `li` elements inside the header also start out with a font size of `24px`. Then in the `li` element's CSS rule, the font size is `0.75em`. This is relative to the element's current font size of `24px`, so the actual font size of the `li` element would be `24px` * 0.75 = `18px`. Finally, the `a` elements inside the `li` elements have a font size of `0.5em`, which is relative to the `li` element's font size of `18px`, so its font size would be `18px` * 0.5 = `9px`.

Due to this cascading effect, sometimes using `em` units for nested elements can cause unintended effects to properties such as font size, padding, and margin.

The `rem` Unit

The `rem` unit is also a relative unit. It's short for "root em" and is relative to the page's base font size. For example, if the base font size is `16px` (remember that this usually doesn't correspond to physical device pixels), a size of `1rem` is equal to `16px`. If you used a size of `1.5rem`, the size would be `16px` * 1.5 = `24px`.

`rem` units are a good choice, especially for layout properties, since the size of `1rem` remains constant throughout the document (unlike the `em` unit). If the browser is zoomed, everything resizes nicely because it's all proportional to the base font size. Because `rem` units are proportional to the base font size, there is no cascading effect like there is with `em` units.

Viewport Units: `vw` and `vh`

The viewport is the area of the page that is currently visible in your web browser. CSS also has units that are relative to the viewport size: `vw` (viewport width) and `vh` (viewport height). Each of these units represents 1% of the viewport size in that direction, that is, `1vw` is 1% of the viewport width and `1vh` is 1% of the viewport height. For example, if the viewport was 1920 pixels wide, a width of `50vw` would be 960 pixels.

If the viewport is resized, then any elements using `vw` or `vh` units will have their sizes adjusted accordingly. Because `vw` and `vh` are relative to the viewport size, they are a good choice when using responsive design techniques.

The % Unit

The `%` unit is relative to the size of another value. What exactly this is relative to depends on the CSS property. For example, for the `font-size` property, the `%` unit is defined as a percentage of the parent element's font size. However, for the `padding` property, `%` is defined as a percentage of the element's width.

Other Units

We have seen some of the most common CSS units, but there are others as well. There are absolute units such as `cm` (centimeters), `mm` (millimeters), `in` (inches), and `pt` (points). These units are sometimes used for print stylesheets but are not often seen in screen stylesheets.

No Units

Some property values take no units at all, but rather just a number. For example, the `opacity` property expects a number between 0 and 1. Another example of this is some flexbox properties such as `flex-grow` and `flex-shrink`, which expect integer numbers without units.

Functions

CSS includes some helpful built-in functions. These functions are used with the values of some CSS properties.

The `calc` Function

The `calc` function combines the different units we covered in the previous section to calculate an exact amount. It can be used anywhere a value is expected. The real power of the `calc` function is that you can have mixed units in the calculation.

For example, suppose you want the height of an element to be 10 pixels short of `1.5rem`. This can easily be accomplished with the `calc` function: `calc(1.5rem - 10px)`. This can be easier than doing the size calculations yourself to specify an exact pixel value. An example of this is shown in Listing 3-14.

Listing 3-14. Using the `calc` function

```
.hero {
  height: calc(1.5rem - 10px);
}
```

Another very useful feature of the `calc` function is that it also works with CSS custom properties or variables. We will look at variables later in this chapter, but Listing 3-15 has an example.

Listing 3-15. Using CSS custom properties with the `calc` function

```
:root {
  --spacing: 0.5rem
}

.container {
  padding: calc(var(--spacing) * 2);
}
```

In the above example, we establish a standard unit of spacing for the document as a variable and can reference that later in the call to the `calc` function. You might also notice the `var` function, which, as we will see later, is used to reference variables.

The `min` and `max` Functions

The `min` and `max` functions take a comma-separated list of other values and returns either the smallest (with `min`) or largest (with `max`) of them.

Suppose you want to make an element take up 50% of the viewport width, but you don't want it to exceed 350 pixels. Listing 3-16 shows how that can be done with the `min` function.

Listing 3-16. Using the `min` function to limit an element's width

```
.content {
  width: min(50vw, 350px);
}
```

The element's width grows with the viewport width, but once the viewport exceeds 700 pixels, the `350px` value becomes the minimum, and that is used. You can use the `min` function to enforce a maximum value, and the `max` function to enforce a minimum value.

There are several other useful CSS functions, which later chapters will cover in the relevant sections.

Colors

One of the most common styles applied with CSS is color. This can include background color, text color, border color, and more. There are many different possible colors, and they can be expressed in multiple ways.

Named Colors

CSS has many predefined color values. They range from basics like `red` and `green` to other shades like `tomato`, `orangered`, and `rebeccapurple`. A full list of these colors can be found at `https://developer.mozilla.org/en-US/docs/Web/CSS/named-color`.

RGB Colors

One common way to define a color is by using the values of its red, green, and blue components (RGB). Any color can be expressed as a combination of RGB values. Each value of red, green, and blue is expressed as a number between 0 and 255.

RGB colors can be specified in several ways. The first is by using a hexadecimal value. The red, green, and blue values are each converted to two hexadecimal digits. These digits are used in RGB order, preceded by a pound sign (#). The hex values can be specified with uppercase or lowercase letters. Table 3-1 shows a few examples of RGB hex notation.

Table 3-1. *Example colors and their hex values*

Color	Hex Code
Black	#000000
White	#FFFFFF
Red	#FF0000
Green	#00FF00
Blue	#0000FF

If each pair of hex digits in the color is the same, you can use a shorthand hex syntax. You can substitute a single digit for each pair of digits. For example, `#00FF00` becomes `#0F0`.

The other way to specify an RGB color is by using the `rgb` function. Instead of hex digits, the red, green, and blue components of the color are specified as base-10 numbers between 0 and 255, or a percentage between 0% and 100%. Table 3-2 shows the usage of the `rgb` function.

Table 3-2. *Example colors using the* `rgb` *function*

Color	RGB Notation
Black	`rgb(0, 0, 0)`
White	`rgb(255, 255, 255)`
Red	`rgb(255, 0, 0)`
Green	`rgb(0, 255, 0)`
Blue	`rgb(0, 0, 255)`

Specifying an Alpha Value

RGB colors can also specify an alpha value, which determines the opacity of the color. The alpha is a value between 0 (fully transparent) and 1 (fully opaque), or a percentage between 0% and 100%. To specify an alpha value, the `rgba` function is used. For example, for pure red with 50% opacity, the color would be defined as `rgba(255, 0, 0, 0.5)`.

The alpha value can also be specified when using hex color codes. Convert the desired alpha value to hexadecimal and append it onto the end of an RGB hex string. In this case, pure red with 50% opacity would be `#FF000080`.

HSL Colors

A color can also be expressed as a combination of its hue, saturation, and lightness values. Hue is specified as a degree of an angle on the color wheel (from 0 to 360 degrees). 0 degrees is red, 120 degrees is green, and 240 degrees is blue.

Saturation is a percentage value of how much color is applied. 0% saturation is a shade of gray, and 100% saturation is the full color from the color wheel. Finally, lightness is also a percentage value. 0% lightness is pure black, and 100% is pure white.

An HSL color is specified using the `hsl` function. The color with a hue of 120 degrees, a saturation of 50%, and a lightness of 50% would be specified as `hsl(120, 50%, 50%)`.

Like RGB colors, HSL colors can also have an alpha value, specified using the `hsla` function. As described earlier, the alpha value can be a number between 0 and 1, or a percentage between 0% and 100%. For the color in the previous example to have 75% opacity, it would be specified as `hsla(120, 50%, 50%, 0.75)`.

Transparent

Anywhere a color is expected, the `transparent` keyword can be used. This applies no color.

Newer Color Syntax

The usage of the `rgb`/`rgba` and `hsl`/`hsla` functions as described above has been the standard for a long time. There is a newer, alternative syntax for these functions that are slightly different.

With this new syntax, you omit the commas between the RGB or HSL numbers. They become separated by spaces, and if you want to specify an alpha value, it's separated from the three color values with a forward slash character (`/`). If you're using an alpha value, you still can use the `rgb` or `hsl` function. The `rgba` and `hsla` functions aren't needed with this new syntax.

Table 3-3 shows some comparisons between the old and new syntax.

Table 3-3. *Comparison of old and new color syntax*

Old Syntax	Equivalent New Syntax
`rgb(0, 0, 0)`	`rgb(0 0 0)`
`rgba(255, 0, 0, 0.5)`	`rgb(255 0 0 / 0.5)`
`hsl(120, 50%, 50%)`	`hsl(120 50% 50%)`
`hsla(120, 50%, 50%, 0.75)`	`hsl(120 50% 50% / 0.75)`

Caution This newer color syntax is not supported in some older browsers.

Color Schemes

Many modern websites have a dark mode, which uses light text on a dark background. You might choose to implement this with a toggle button, but you can also automatically determine if the user has enabled dark mode in their operating system settings (where supported).

You can do this by using the `prefers-color-scheme` *media query*. We haven't covered media queries yet, but they use a special at-rule, `@media`, and specify a query about the device's configuration, capabilities, or screen size. The media query contains CSS rules that are only applied when the media query matches.

We'll explore media queries in more detail when we cover responsive design techniques in Chapter 12, but Listing 3-17 has an example of using the `prefers-color-scheme` media query to automatically apply dark mode styling.

Listing 3-17. Using `prefers-color-scheme` to apply dark mode automatically

```
.header {
  background: white;
  color: black;
}
```

```
@media (prefers-color-scheme: dark) {
  .header {
    background: black;
    color: white;
  }
}
```

By default, the header has a white background and black text. If the user has enabled dark mode in their operating system settings, the media query matches and applies the rule it contains. This sets the header's background to black and text color to white.

When using dark mode, you'll also want to specify the `color-scheme: dark` property. Without this, some UI elements such as scroll bars are still rendered in a light color. You should apply this property to the root element, ensuring that all elements on the page are rendered with a dark color scheme. This is shown in Listing 3-18.

Listing 3-18. Using a dark color scheme

```
@media (prefers-color-scheme: dark) {
  :root {
    color-scheme: dark;
  }
}
```

As an alternative to using the media query, you can also use the `light-dark` function. To use this function, you first need to set `color-scheme` to the special value `light dark`. Once this is set, you can use `light-dark` wherever a color is expected. The first argument is the color to use for a light color scheme, and the second is the color to use for a dark color scheme.

Listing 3-19 shows the example from Listing 3-18, adapted to use the `light-dark` function.

Listing 3-19. Using the `light-dark` function

```
:root {
  /* This is required for the light-dark function to work */
  color-scheme: light dark;
}
```

```
.header {
  background: light-dark(white, black);
  color: light-dark(black, white);
}
```

As before, the background and text colors change depending on the operating system's color scheme setting.

Overflow

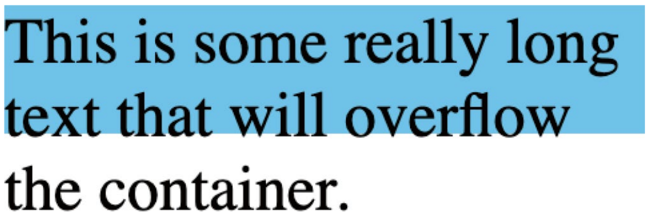
As discussed earlier, every HTML element is a rectangular box. Normally, the size of an element expands to fit its content, as shown in some earlier examples. However, what happens when an explicit height is set, and the content does not fit inside the element's dimensions? This is a condition known as *overflow*. Listing 3-20 contains an example showing text overflowing its container.

Listing 3-20. Demonstrating overflow of an element's content

```
<style>
  .container {
    background-color: skyblue;
    height: 2rem;
    width: 10rem;
  }
</style>

<div class="container">
  This is some really long text that will overflow the container.
</div>
```

The overflow can be seen in Figure 3-14.



This is some really long text that will overflow the container.

Figure 3-14. *Text overflowing its container*

The text content is too long to fit in a `2rem` by `10rem` container, so it overflows the container element. Note that the content only overflows vertically. This is because the default behavior is to wrap the text to the next line when it doesn't fit on one line. This ensures it doesn't overflow horizontally, but because an explicit height is set, the container does not grow to fit its content.

If you insert a fixed width element inside the container, as shown in Listing 3-21, then the content overflows both horizontally and vertically.

Listing 3-21. Adding a fixed width element inside the container

```
<style>
  .container {
    background-color: skyblue;
    height: 2rem;
    width: 10rem;
  }

  .container .banner {
    background: #999999;
    width: 15rem;
  }
</style>

<div class="container">
  <div class="banner">This is a banner</div>
  This is some really long text that will overflow the container.
</div>
```

As Figure 3-15 shows, there is now horizontal and vertical overflow. The banner element has an explicit width set, larger than its container's width, so it overflows horizontally. The long text content overflows vertically as it did before.

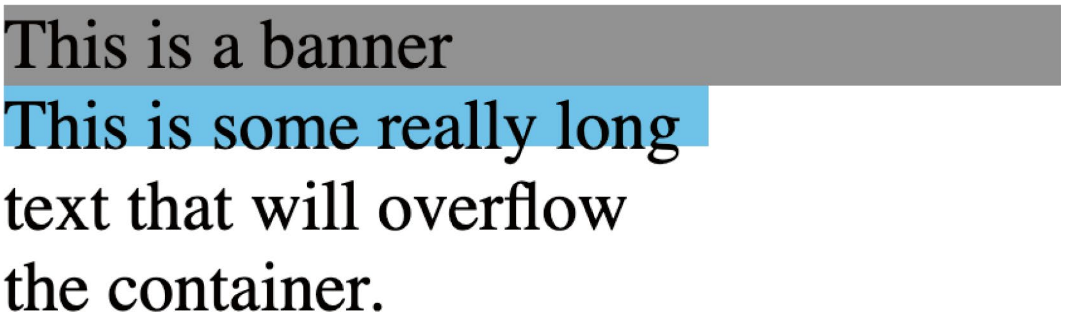

This is a banner
This is some really long text that will overflow the container.

Figure 3-15. *Horizontal and vertical overflow*

Handling Overflow

You have some control over how overflow is handled with the `overflow` property. This property handles both horizontal and vertical overflow together. They can also be handled independently with the `overflow-x` and `overflow-y` properties. The default value is `visible`, which results in the behavior from the previous examples. There are a few other options available for handling overflow.

Using `overflow: hidden`

When the `overflow` property is set to `hidden`, the overflowing content is simply not displayed. It's clipped by the bounds of the containing element. Listing 3-22 shows an example of using `overflow: hidden`.

Listing 3-22. Hiding overflowing content with `overflow: hidden`

```
<style>
  .container {
    background-color: skyblue;
    height: 2rem;
    width: 10rem;
    overflow: hidden;
  }
  .container .banner {
    background: #999999;
    width: 15rem;
  }
</style>

<div class="container">
  <div class="banner">This is a banner</div>
  This is some really long text that will overflow the container.
</div>
```

As Figure 3-16 shows, the overflowing content is clipped by the bounds of the container element. The portion that overflows is not visible.

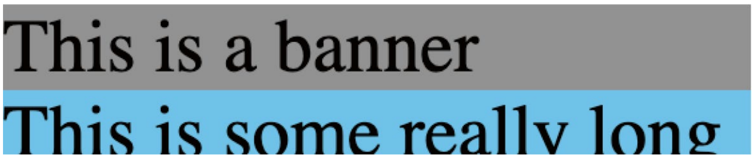


Figure 3-16. *The overflowing content is clipped with `overflow: hidden`.*

Using `overflow: scroll` and `overflow: auto`

When `overflow` is set to `scroll`, the overflowing content is initially not visible. However, scrollbars are provided so that the user can scroll the containing element to view the overflowing content. The scrollbars are always provided, even if the content does not overflow. See Listing 3-23 for an example of using `overflow: scroll`.

Listing 3-23. Using `overflow: scroll`

```
<style>
  .container {
    background-color: skyblue;
    height: 5rem;
    width: 10rem;
    overflow: auto;
  }
  .container .banner {
    background: #999999;
    width: 15rem;
  }
</style>

<div class="container">
  <div class="banner">This is a banner</div>
  This is some really long text that will overflow the container.
</div>
```

Figure 3-17 shows the rendered content with scrollbars.

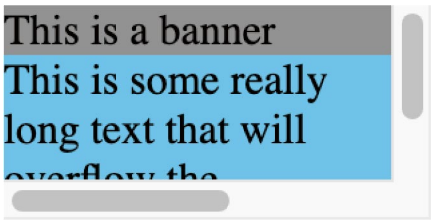

This is a banner
This is some really
long text that will
overflow the

Figure 3-17. *Showing scrollbars with `overflow: scroll`*

If the content does not overflow its container, overflow: scroll still renders scrollbars, as shown in Figure 3-18.

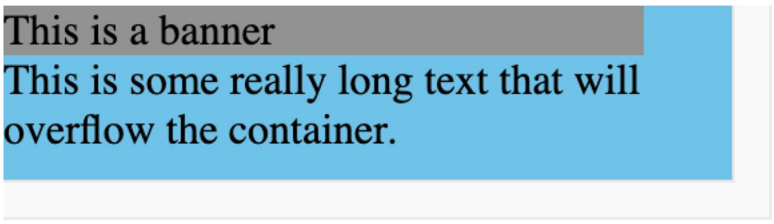

This is a banner
This is some really long text that will
overflow the container.

Figure 3-18. *Empty scrollbars are still rendered*

If you want to avoid these empty scrollbars when the content isn't overflowing, you can use `overflow: auto`. This works the same as `overflow: scroll`, except scrollbars are only shown if they are needed.

CSS Variables

Variables are a common feature in all programming languages. A variable is a way to store a piece of data, under a descriptive name, and that value can be referenced later by the variable name.

There are many useful reasons to use CSS variables (officially known as CSS custom properties). Suppose you're designing a website for your company. Your brand color, `#3FA2D9`, is used in many places throughout the site's CSS. If, later, the site's brand color changes, you'll now have to change every instance of the brand color where you used `#3FA2D9`.

Instead, you can define a `brand-color` variable, and reference that variable everywhere you need the brand color. Later, if you need to change the brand color, you need only change the `brand-color` variable, and it is updated everywhere it's referenced.

Using Variables

Variables are declared with two dashes followed by the variable name. Later, to reference a variable's value, you pass the variable name to the `var` function.

A variable can be declared on any element. Variables are then inherited by descendant elements. Variables are often declared on the document's root element, so that they are inherited by the entire document, using the special `:root` selector.

Listing 3-24 shows how you can use a `brand-color` variable.

Listing 3-24. Using a CSS variable

```
<style>
  :root {
    --brand-color: #3FA2D9;
  }

  .site-header {
    background: var(--brand-color);
  }
</style>

<header class="site-header">
  Site Header
</header>
```

The header's background color uses the value of the `brand-color` variable, `#3FA2D9`.

The `var` function also takes an optional second argument, which is a fallback value to use in case the referenced variable name isn't defined, as demonstrated in Listing 3-25.

Listing 3-25. Using a fallback value for a CSS variable

```
.site-header {
  background: var(--brand-color, blue);
}
```

If the `brand-color` variable isn't defined, the background is set to `blue`. Variables can also reference other variables, as shown in Listing 3-26.

Listing 3-26. Referencing variables from other variables

```
<style>
  :root {
    --primary-border-color: red;
    --primary-border-style: solid;
    --primary-border-width: 3px;
    --primary-border:
      var(--primary-border-width)
      var(--primary-border-style)
      var(--primary-border-color);
  }

  .container {
    border: var(--primary-border);
    width: 10rem;
  }
</style>

<div class="container">Hello World!</div>
```

The `primary-border` variable incorporates the other three variables, and that variable is referenced in the container's style, which applies a `3px solid red` border, as shown in Figure 3-19.

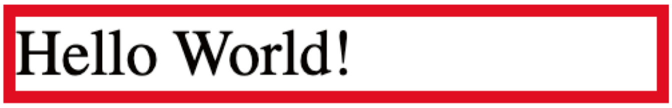


Figure 3-19. *The result showing a border calculated from multiple CSS variables*

Using Variables with the `calc` Function

You can even reference variables when using the `calc` function. Consider the example in Listing 3-27. There is a container with six rows. Suppose you want the container to be tall enough to show three visible rows, and the rest should overflow and be accessed by scrolling.

Listing 3-27. Using variables for layout

```
<style>
:root {
    --row-height: 1.5rem;
    --visible-rows: 3;
  }

  .container {
    border: 1px solid red;
    height: calc(var(--row-height) * var(--visible-rows));
    overflow: auto;
    width: 10rem;
  }

  .row {
    line-height: var(--row-height);
  }
</style>

<div class="container">
  <div class="row">Row one</div>
  <div class="row">Row two</div>
  <div class="row">Row three</div>
  <div class="row">Row four</div>
  <div class="row">Row five</div>
  <div class="row">Row six</div>
</div>
```

The result is shown in Figure 3-20. The first three rows are visible, and the rest are accessible with the scrollbar.

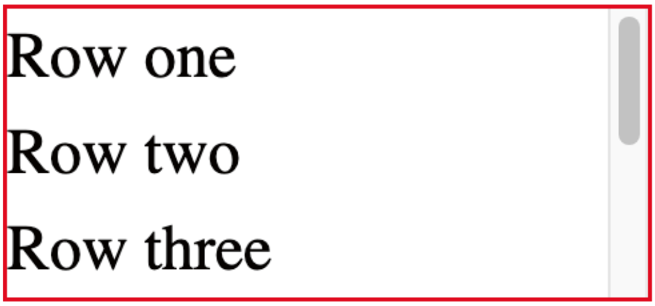

Row one
Row two
Row three

Figure 3-20. *Three visible rows*

This example sets two variables: an explicit row height of `1.5rem` and a visible row count of 3. The container height should exactly equal three rows, so in the container we use `calc` to multiply the `row-height` variable by the `visible-rows` variable, which yields the correct height.

The rest of the rows overflow the container. Since we're using `overflow: auto`, a scrollbar is shown to access the overflowing rows.

Summary

- All elements are represented by a box with a content area, padding, border, and margin.
- There are three main types of elements: block, inline, and inline-block.
- There are many different units for CSS values:
  - `px` refers to logical pixels.
  - `em` is relative to the element's font size.

    - `rem` is relative to the document's base font size.
    - `vw` and `vh` are relative to the viewport size.
- The calc function is used to compute CSS values from multiple other values, potentially with different units.
- Colors can be defined in several ways:
    - Named colors: `red`, `blue`, `orangered`
    - Hexadecimal RGB or RGBA: `#FF0000` or `#FF000080`
    - The `rgb` function: `rgb(255, 0, 0)` or `rgb(255 0 0)`
    - The `hsl` function: `hsl(90, 50%, 25%)` or `hsl(90 50% 25%)`
- You can use the `prefers-color-scheme` media query to detect the operating system's dark mode setting.
- If an element's content can't fit inside of it, the content overflows.
- Overflow handling can be changed with the `overflow`, `overflow-x`, and `overflow-y` properties.
- A variable is declared with two leading slashes: `--var-name`.
- A variable is referenced with the `var` function: `var(--var-name)`.
- Variables are inherited by descendant elements.

CHAPTER 4

Basic Styling

By now, you have a solid grasp of the main underlying concepts of CSS. Now it's time to dive in and start learning some CSS properties and styling techniques. We'll start with the basics in this chapter.

Property Values

Before we start exploring specific CSS properties, here are a few notes on certain property values.

Global Keywords

Most CSS properties support several global keywords in their values, each with a special meaning. These include

- `initial`: Uses the initial value set by the browser's built-in stylesheet.
- `inherit`: Takes the value used by the element's parent.
- `unset`: If the property naturally inherits from its parent, such as `font-size`, it is set to the inherited value. Otherwise, it is set to the initial value from the browser's stylesheet.

Shorthand and Multiple Values

You've seen how many CSS properties, such as `border-width`, `padding`, and `margin`, can take a single value. These are known as shorthand properties. For these properties, the single value given is used for the top, bottom, left, and right.

Each shorthand property has four equivalent properties for each side of the element. For example, for the `padding` shorthand element, there are also `padding-top`, `padding-bottom`, `padding-left`, and `padding-right` properties.

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_4

If you want to specify different values for different sides of the element, you can still use the shorthand property – just give it multiple values.

If one value is given, as we've seen, it applies to all four sides of the element. Listing 4-1 shows an example of this using the `border-width` property.

Listing 4-1. Using a single value for the `border-width` property

```
<style>
  .container {
    border-color: red;
    border-style: solid;
    border-width: 1px;
  }
</style>

<div class="container">Hello world!</div>
```

Figure 4-1 shows that all four borders have the same width.

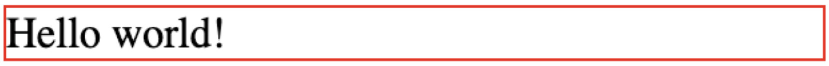


Figure 4-1. *All four borders have the same width.*

If two values are specified, the first applies to the top and bottom, and the second applies to the left and right. An example of this is seen in Listing 4-2.

Listing 4-2. Two values for the `border-width` property

```
<style>
  .container {
    border-color: red;
    border-style: solid;
    border-width: 1px 5px;
  }
</style>

<div class="container">Hello world!</div>
```

Figure 4-2 shows the different border widths. Note the different border widths in Figure 4-2 on the top and bottom compared to the left and right.

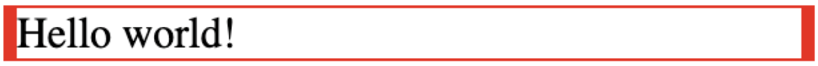


Figure 4-2. *The rendered result showing the different border widths*

If three values are specified, the first applies to the top, the second applies to the left and right, and the third applies to the bottom. Listing 4-3 has an example of this.

Listing 4-3. Three values for the `border-width` property

```
<style>
  .container {
    border-color: red;
    border-style: solid;
    border-width: 1px 5px 10px;
  }
</style>

<div class="container">Hello world!</div>
```

Figure 4-3 shows the resulting border widths.

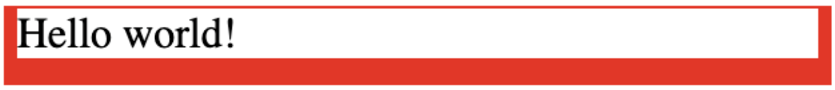


Figure 4-3. *The rendered result showing the three different border widths*

Finally, if four values are specified as shown in Listing 4-4, they are applied in clockwise order, starting at the top.

Listing 4-4. Four values for the `border-width` property

```
<style>
  .container {
    border-color: red;
    border-style: solid;
    border-width: 1px 5px 10px 20px;
  }
</style>

<div class="container">Hello world!</div>
```

Figure 4-4 shows the border widths applied in a clockwise direction.

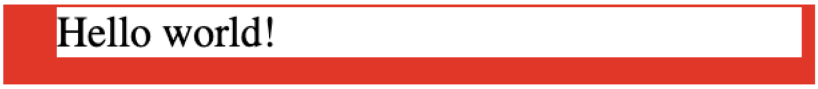


***Figure 4-4.** The borders are applied clockwise, starting with the top*

Borders

For most elements, there is no border by default. CSS defines several properties for styling borders.

Setting the Border Color

The color of the border is set with the `border-color` property.

Setting the Border Width

The `border-width` property determines how thick the border is. The value of `border-width` can be a value like `3px`. There are also some predefined values: `thin`, `medium`, and `thick`, as shown in Figure 4-5.

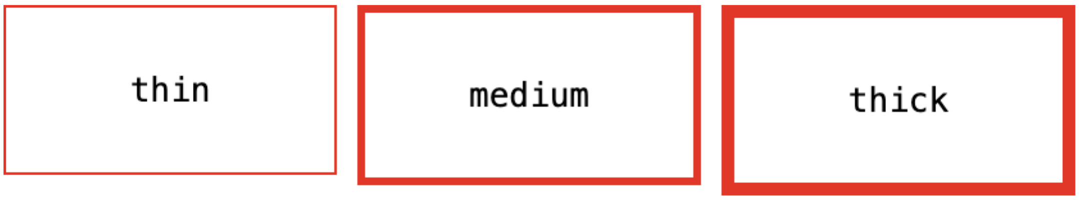


***Figure 4-5.** Predefined border width values*

Setting the Border Style

The `border-style` property determines the visual appearance of the border. In addition to `none` (the default), there are several available border styles, as shown in Figure 4-6.

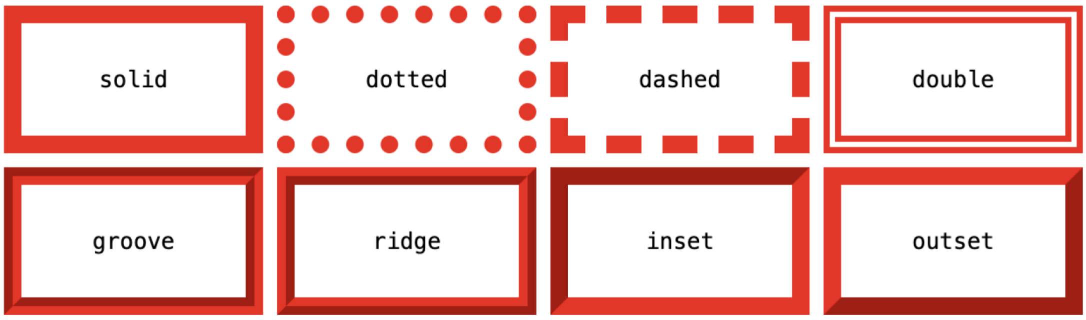


Figure 4-6. *Border styles*

Setting All Properties with the Shorthand

The three above properties can be combined into a single property with the `border` shorthand property, as shown in Listing 4-5. The values should be specified in the order `<width> <style> <color>`.

Listing 4-5. The `border` shorthand property

```
.container {
  border: 5px solid red;
}
```

The code in Listing 4-6 sets a `border-width` of `5px`, a `border-style` of `solid`, and a `border-color` of `red`.

Specifying Border Collapse

The `border-collapse` property only applies to `table` elements. It controls how borders are preserved or collapsed between adjoining table cells. The default value is `separate`. With this default behavior, a table's borders are not combined. Consider the table in Figure 4-7, where each cell has a different colored border.

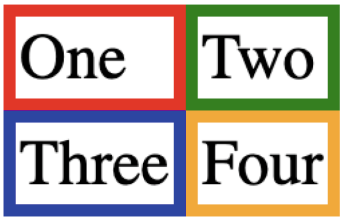


Figure 4-7. *Borders are not collapsed*

Notice that each cell's border is seen in its entirety – the borders do not collapse. Figure 4-8 shows the result if we set `border-collapse` to `collapse`. Any cell borders adjacent to another border have been collapsed into a single border.

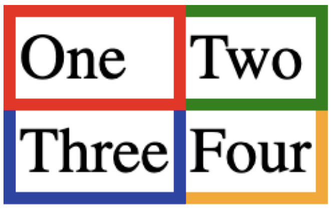


Figure 4-8. *Borders are now collapsed*

Setting a Border Radius for Rounded Corners

By default, elements have 90-degree rectangular corners. That isn't always the most aesthetically pleasing design, though. To address this, CSS has the `border-radius` property. This property gives the element rounded corners. The corners can be circular or elliptical. A simple example is shown in Listing 4-6.

Listing 4-6. Setting a border radius

```
<style>
  .rounded-corners {
    background-color: red;
    border-radius: 10px;
    height: 5rem;
    width: 5rem;
  }
</style>

<div class="rounded-corners"></div>
```

This creates rounded corners with a radius of `10px`, as shown in Figure 4-9.

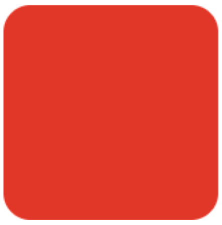


Figure 4-9. *A box with rounded corners*

What does it mean for the corner to have a radius of `10px`? Imagine a circle drawn over each corner. That circle's radius is `10px`. This visualization is shown in Figure 4-10.

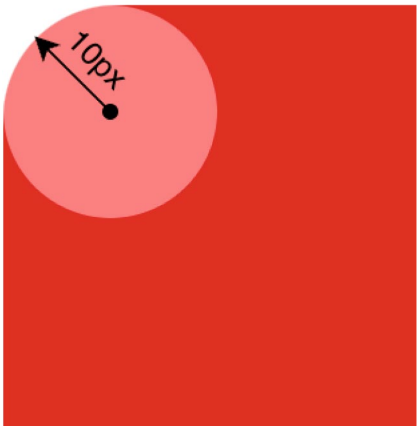


Figure 4-10. *Border radius visualized*

The `border-radius` can be elliptical as well as circular. Each of the two radii of the ellipse is specified, separated by a slash, as shown in Listing 4-7.

Listing 4-7. Using an elliptical `border-radius`

```
<style>
  .rounded-corners {
    background-color: red;
    border-radius: 20px / 10px;
    height: 5rem;
    width: 5rem;
  }
</style>

<div class="rounded-corners"></div>
```

Figure 4-11 shows the resulting corners.

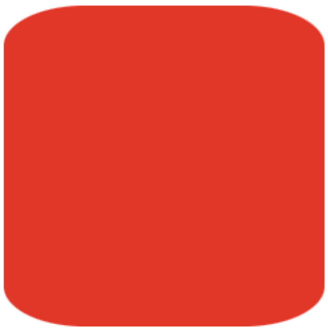


Figure 4-11. *A box with an elliptical border radius*

Similarly to the circular radius, the elliptical radius effect is applied with an ellipse in each corner. The horizontal radius is given first, followed by the vertical. This is visualized in Figure 4-12.

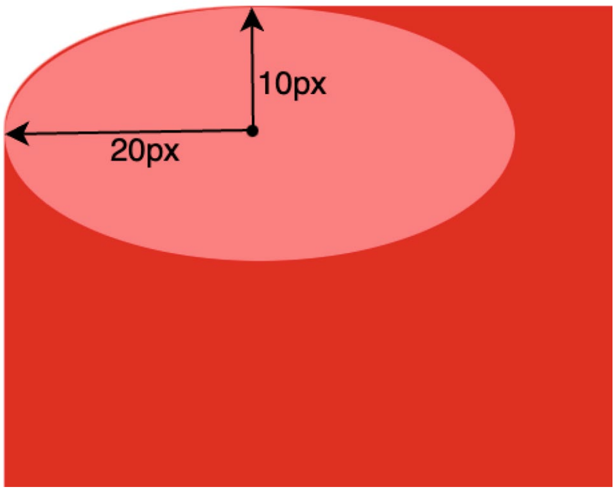


Figure 4-12. *Visualizing the elliptical border radius*

You can also specify a different `border-radius` for each corner, creating some interesting shapes. Note that when you specify `border-radius` in this way, an elliptical border-radius does not have a slash separating the horizontal and vertical radii. An example of this is shown in Listing 4-8.

Listing 4-8. Specifying different border radius properties

```
<style>
  .rounded-corners {
    background-color: red;
    border-bottom-right-radius: 10px 20px;
    border-bottom-left-radius: 5px;
    border-top-left-radius: 20px 10px;
    border-top-right-radius: 50%;
    height: 5rem;
    width: 5rem;
  }
</style>

<div class="rounded-corners"></div>
```

The resulting shape is shown in Figure 4-13.

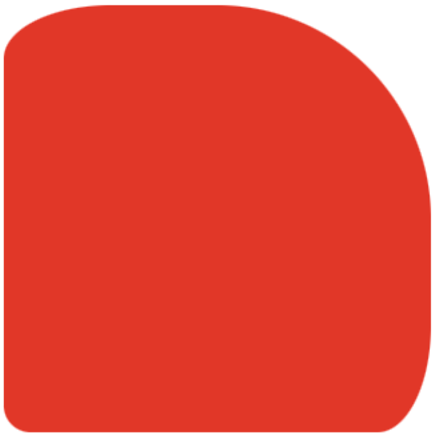


Figure 4-13. *A shape using different border-radius values for each corner*

Box Shadows

Elements can also have a box shadow. As the name implies, this is a shadow applied to the element's box, appearing just outside the border area. The shadow follows the shape of the box. If there are rounded corners applied with border-radius, the box shadow also has the rounded corners.

The box shadow is controlled by the `box-shadow` property. A box shadow has a color, and its dimensions can be specified with up to four values, which are

- X offset
- Y offset
- *Blur radius*: How far out the shadow is blurred
- *Spread radius*: How far the shadow extends beyond the element's dimensions

The box shadow is the same size as the element, and when the X and Y offsets are zero, it appears directly behind the element. In this case, the shadow won't be visible without a blur or spread radius.

At a minimum, the X and Y offsets must be given. By default, the blur and spread radius are zero. Listing 4-9 has a straightforward example of a box shadow.

Listing 4-9. A simple box shadow

```
<style>
  .shadow {
    box-shadow: 5px 5px black;
    background: #CCCCCC;
    width: 10rem;
    height: 5rem;
  }
</style>

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

This applies a box shadow offset by 5 pixels in the horizontal and vertical directions and with no blur or spread radius. The result is shown in Figure 4-14.

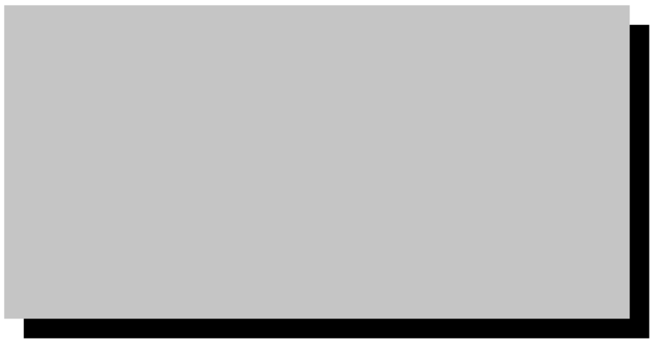


***Figure 4-14.** A simple offset box shadow*

In Listing 4-10, we apply a blur radius to this shadow of 10 pixels.

Listing 4-10. Adding a blur radius

```
<style>
  .shadow {
    box-shadow: 5px 5px 10px black;
    background: #CCCCCC;
    width: 10rem;
    height: 5rem;
  }
</style>

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

This gives the box shadow a blurred look, as shown in Figure 4-15.

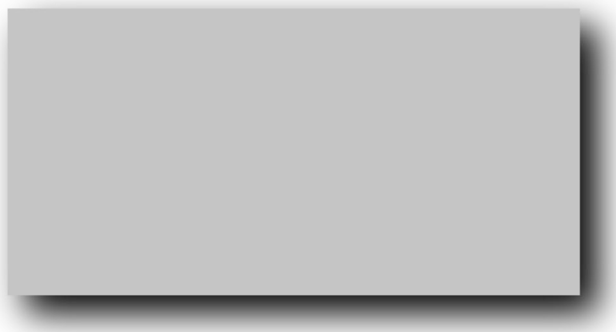


***Figure 4-15.** A shadow with a blur effect*

Finally, we can add a spread radius, as we have done in Listing 4-11.

Listing 4-11. Adding a spread radius

```
<style>
  .shadow {
    box-shadow: 5px 5px 10px 5px black;
    background: #CCCCCC;
    width: 10rem;
    height: 5rem;
  }
</style>
```

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

The results are shown in Figure 4-16.

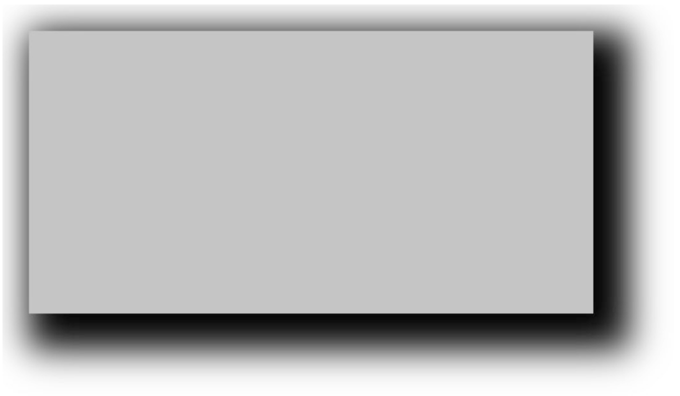


Figure 4-16. *The rendered shadow with a spread radius added*

If the X and Y offsets are set to zero, the shadow spreads and blurs evenly in all directions. Listing 4-12 demonstrates how to do this.

Listing 4-12. Setting the offsets to zero

```
<style>
  .shadow {
    box-shadow: 0 0 10px 5px black;
    background: #CCCCCC;
    width: 10rem;
    height: 5rem;
  }
</style>

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

The even shadow is shown in Figure 4-17.

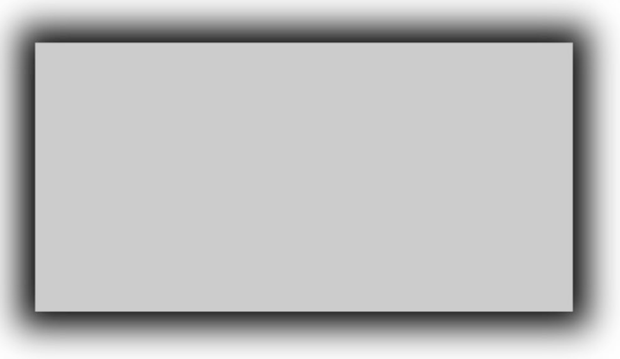


Figure 4-17. *An evenly distributed box shadow*

Box shadows can also be inside the element instead of behind it. To do this, add the `inset` keyword to the `box-shadow` value. An inset box shadow starts at the element's border and is drawn inward. Listing 4-13 has an example of an inset box shadow.

Listing 4-13. An inset box shadow

```
<style>
  .shadow {
    box-shadow: 0 0 25px black inset;
    background: #CCCCCC;
    width: 10rem;
    height: 5rem;
    border: 5px solid red;
  }
</style>

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

Figure 4-18 shows how this inset box shadow is rendered. Notice how it starts inside the red border area.

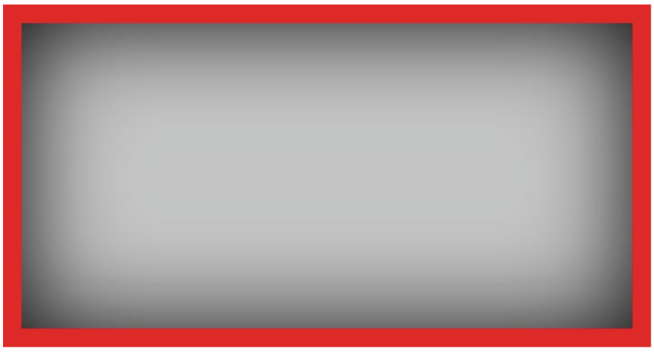


Figure 4-18. *The rendered inset box shadow*

You can also apply multiple box shadows to a single element. The `box-shadow` property accepts multiple box shadow definitions, separated by a comma, as shown in Listing 4-14.

Listing 4-14. Applying multiple box shadows

```
<style>
  .shadow {
    box-shadow: 0 0 15px 5px black, 0 0 5px 5px red;
    background: #CCCCCC;
    width: 10rem;
    height: 5rem;
  }
</style>

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

This applies a black box shadow and a red one, as shown in Figure 4-19.

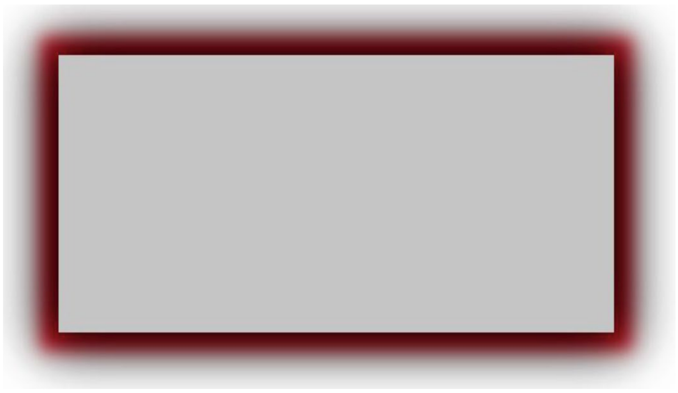


***Figure 4-19.** Multiple box shadows on a single element*

Finally, you can also apply an outer box shadow and an inset one together to create an inner and outer shadow. An example of this is shown in Listing 4-15.

Listing 4-15. Applying outer and inset shadows

```
<style>
  .shadow {
    box-shadow: 0 0 10px 0 black, 0 0 25px red inset;
    background: #CCCCCC;
    width: 10rem;
    height: 5rem;
  }
</style>

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

The result is shown in Figure 4-20.

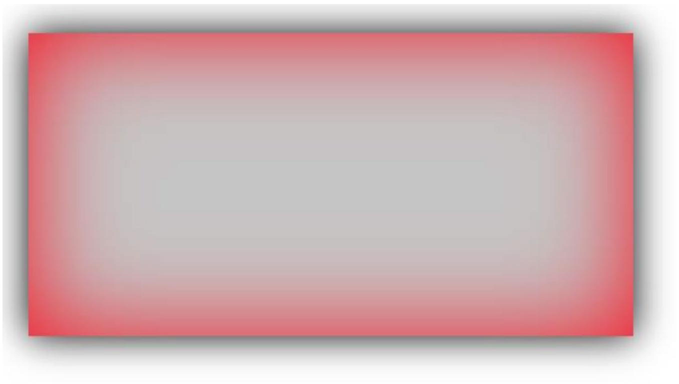


Figure 4-20. *A red inner shadow with a black outer shadow*

Opacity

By default, most elements start out with a transparent background. When a background color or image is assigned, that element becomes opaque. You cannot see through the element to what's behind it. Borders and text are also opaque.

You can change this behavior with the `opacity` property. Opacity applies to the entire element – background, border, text, images, and any other content within that element or its children.

The `opacity` property takes a number between `0` and `1` or a percentage from `0%` to `100%`. This sets the level of transparency of the element. An opacity of `0.5`, or `50%`, is half transparent.

Consider the example in Listing 4-16, an outer `div` element with another `div` inside of it. Note the different background colors and how the inner element is fully opaque – that is, it covers the background of the outer element.

Listing 4-16. One element inside another

```
<style>
  .outer {
    background: red;
    height: 10rem;
    width: 10rem;
  }
  .inner {
    background: blue;
    color: white;
    height: 8rem;
    width: 8rem;
  }
</style>

<div class="outer">
  <div class="inner">
  </div>
</div>
```

Figure 4-21 shows the rendered result.

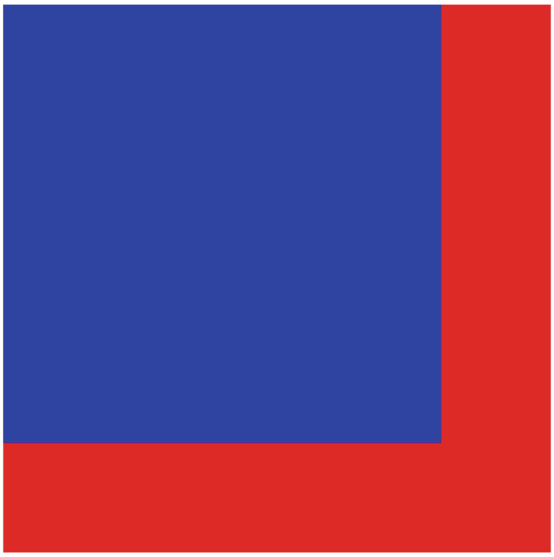


Figure 4-21. *The two elements, fully opaque*

Let's add some text to the inner element and make it partially transparent by setting its `opacity` property to 0.5. This is done in Listing 4-17.

Listing 4-17. Making the inner element partially transparent

```
<style>
  .outer {
    background: red;
    height: 10rem;
    width: 10rem;
  }

  .inner {
    background: blue;
    color: white;
    height: 8rem;
    width: 8rem;
    opacity: 0.5;
  }
</style>

<div class="outer">
  <div class="inner">Inner Element</div>
</div>
```

Figure 4-22 shows the result of this change. Since the inner element is 50% transparent, its blue background mixes with the outer element's red background, and it becomes purple. Notice that the text is partially transparent as well.

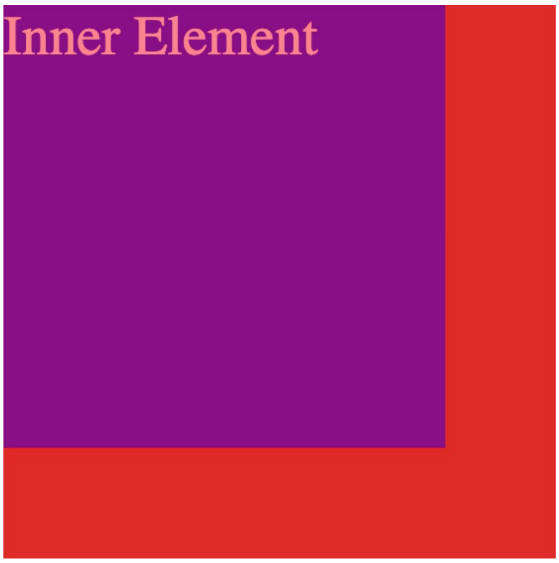


Figure 4-22. *The demo elements, with opacity applied to the inner element*

Setting the `opacity` to `0.2` makes the inner box even more transparent, and barely visible, as shown in Figure 4-23.

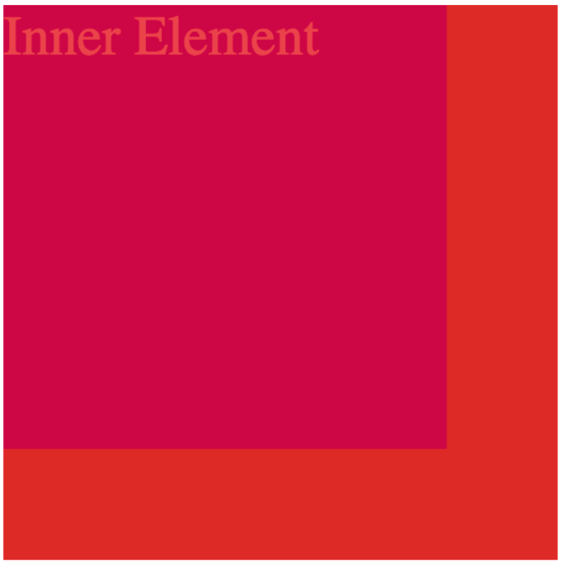


Figure 4-23. *Decreasing the opacity*

Hiding Elements

You sometimes want to have an element in the page that is visually hidden, maybe temporarily. There are a few ways you can hide an element using CSS.

Using `display: none`

We've already seen how the `display` property can make an element either `inline`, `block`, or `inline-block`. When you set the `display` property to `none`, the element is hidden and removed from the flow of the document as if it was never there. It does, however, remain in the DOM. Other elements move to fill in the space.

Consider the three boxes shown in Figure 4-24.

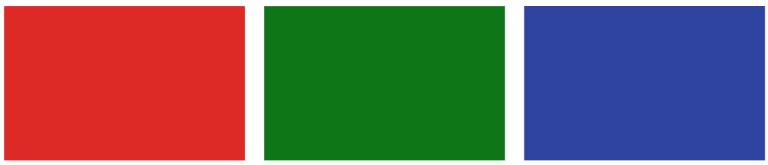


Figure 4-24. *An example layout containing three boxes*

If you set the green middle element's `display` property to `none`, it is removed from the document flow, and the other blocks move closer together to fill the empty space, as shown in Figure 4-25.

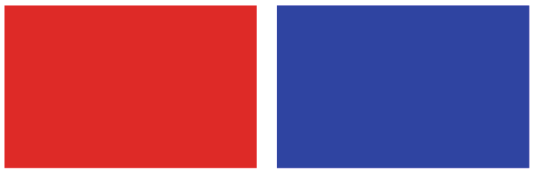


Figure 4-25. *The middle element hidden with `display: none`*

Using `visibility: hidden`

Another way to hide an element is by setting its `visibility` property to `hidden`. This hides the element like `display: none`, but it behaves differently. With `visibility: hidden`, the flow of the document is not affected. The other elements do not move to fill the empty space left by the hidden element. To make a hidden element reappear, set its `visibility` property to `visible`.

Consider the example from Figure 4-23. If you change the middle box to use `visibility: hidden`, it disappears, but an empty space remains, as shown in Figure 4-26.

Figure 4-26. *The empty space left by `visibility: hidden`*

Setting opacity to 0

You can also hide an element by setting its `opacity` property to `0`. This has the same visual effect as `visibility: hidden`. The element is hidden from view, but the layout is unchanged.

One reason to use `opacity: 0` instead of `visibility: hidden` is that you can use a CSS transition to gradually show and hide the element. With `visibility: hidden`, the element appears or disappears immediately. We'll talk more about CSS transitions in Chapter 9.

Outline

Elements can also have an outline, which is outside of their border. This is done using the `outline` property. To illustrate this concept, Listing 4-18 creates an element with a border and an outline.

Listing 4-18. Applying an outline

```
<style>
  .box {
    width: 128px;
    height: 64px;
    border: 5px solid red;
    outline: 5px solid blue;
  }
</style>

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

The result is shown in Figure 4-27.

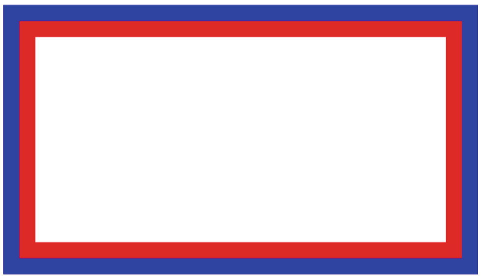


Figure 4-27. *The rendered outline*

The blue outline appears immediately outside of the red border and is flush up against the border. You can add space between the border and the outline with the `outline-offset` property, as shown in Listing 4-19.

Listing 4-19. Applying an outline offset

```
<style>
  .box {
    width: 128px;
    height: 64px;
    border: 5px solid red;
    outline: 5px solid blue;
    outline-offset: 8px;
  }
</style>

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

As Figure 4-28 shows, there is now an 8-pixel offset between the border and the outline.

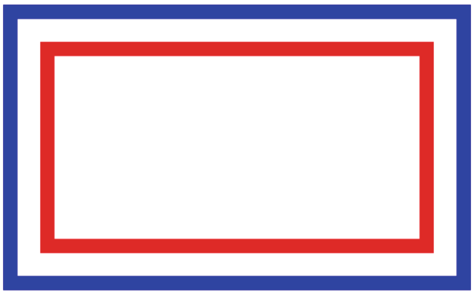


Figure 4-28. *The box with an outline offset*

One key difference between border and outline is that outline does not affect the document's layout. An element's outline can overlap with adjacent elements. Listing 4-20 shows such an example.

Listing 4-20. Two adjacent elements, one with an outline

```
<style>
  .box {
    width: 128px;
    height: 64px;
  }

  .box1 {
      border: 5px solid red;
      outline: 5px solid blue;
      outline-offset: 8px;
    }

  .box2 {
    border: 5px solid green;
  }
</style>

<div>
  <div class="box box1"></div>
  <div class="box box2"></div>
</div>
```

When these elements are rendered, the outline from the red box overlaps the green box, as shown in Figure 4-29.

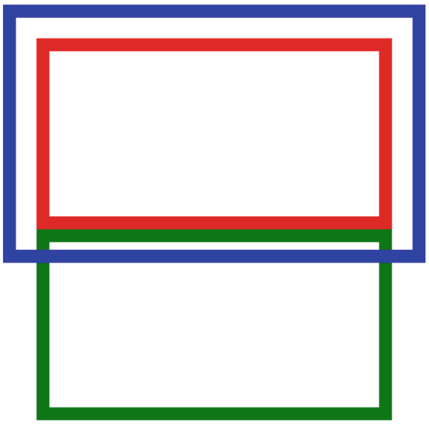


Figure 4-29. *The outline can overlap adjacent elements*

By default, most elements do not have an outline. The main exception to this rule is form input fields. They have an outline when the element has focus. You can override this behavior by setting the `outline` property to `none`. However, this is not recommended as it can create accessibility issues if there is no visual way to tell that an element has focus.

Summary

- A border has a style, width, and color.
- Rounded corners can be created with the `border-radius` property.
- An element can be given an inner and/or outer box shadow with the `box-shadow` property.
- The `opacity` property determines the transparency of an element.
- An element can be hidden by using `display: none`, `visibility: hidden`, or `opacity: 0`.
- An element can have an outline, which appears outside of its border.

CHAPTER 5

Backgrounds and Gradients

Most HTML elements can have a background. Backgrounds can be images, solid colors, or even gradients. Multiple background effects can be combined.

Solid Background Colors

The simplest type of background is a solid color. Solid background colors are applied using the `background-color` property. This accepts any valid CSS color expression, as shown in Listing 5-1.

Listing 5-1. A solid background color

```
<style>
  .red-background {
    background-color: #FA7587;
  }
</style>

<div class="red-background">Hello world!</div>
```

As expected, this applies a red background to the element, as shown in Figure 5-1.

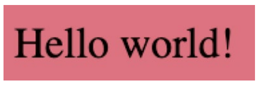


Figure 5-1. *The element with a red background*

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_5

Background Images

Images can also be used as an element's background. There are several properties that control the background image.

Applying a Background Image with the `background-image` Property

One or more background images can be applied using the `background-image` property. This accepts one or more image URLs. If multiple URLs are specified, the background images are stacked on top of each other, with the first image on top.

A background image's URL is specified with the `url` function. It takes one argument, which is a string that can be an absolute or relative URL. Here are a few examples:

- Absolute URL: `url('https://placecats.com/neo/300/200')`
- Relative URL: `url('/header.png')`

Listing 5-2 shows an example of using the `background-image` function.

Listing 5-2. Using a background image

```
<style>
    .cat {
        background-image: url('./cat.jpg');
        height: 200px;
        width: 300px;
    }
</style>

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

Figure 5-2 shows the element with the background image.

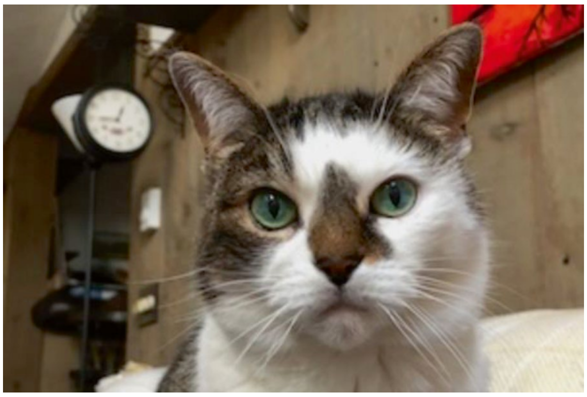


Figure 5-2. *The element with a background image*

Repeating a Background with the `background-repeat` Property

If an element is larger than its background image, by default the image will be repeated to fill the element. Listing 5-3 creates an element with a tiled background image.

Listing 5-3. Using a background image on a larger element

```
<style>
    .cat {
        background-image: url('./small-cat.jpg');
        height: 200px;
        width: 300px;
    }
</style>

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

Figure 5-3 shows the repeated background image (the default behavior).

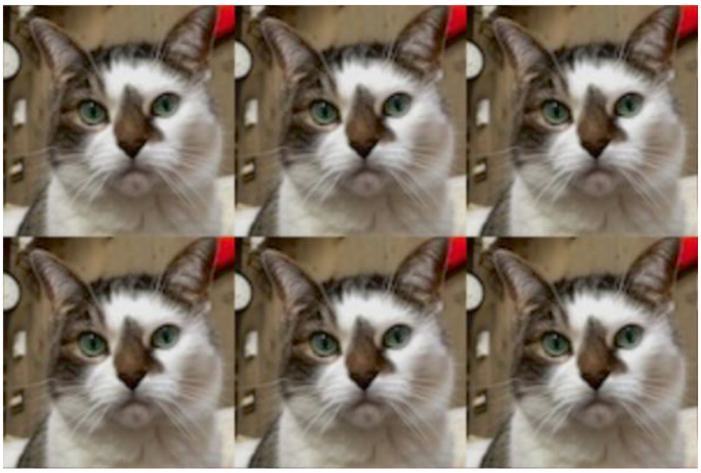


Figure 5-3. *The background repeating itself*

This behavior can be changed with the `background-repeat` property. Repeating can be disabled by setting it to `no-repeat`. If the background isn't repeated to cover the entire element, the background color, if any, will show through. This is demonstrated in Listing 5-4.

Listing 5-4. Disabling background repeat

```
<style>
    .cat {
        background-image: url('./small-cat.jpg');
        background-color: #999999;
        background-repeat: no-repeat;
        height: 200px;
        width: 300px;
    }
</style>

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

The result is shown in Figure 5-4.

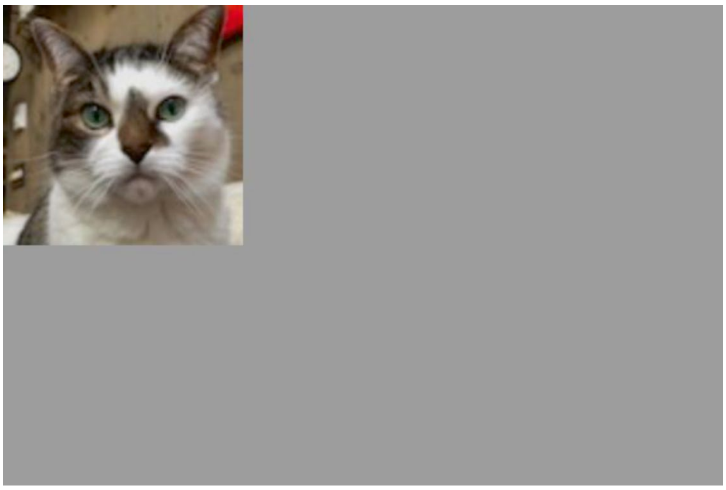


Figure 5-4. *The background is not repeated*

The background image can be repeated just horizontally or just vertically by specifying a `background-repeat` of `repeat-x` or `repeat-y`. Listing 5-5 shows an example of repeating the background vertically only.

Listing 5-5. Repeating the background vertically

```
<style>
    .cat {
        background-image: url('./small-cat.jpg');
        background-color: #999999;
        background-repeat: repeat-y;
        height: 200px;
        width: 300px;
    }
</style>

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

Figure 5-5 shows the result.

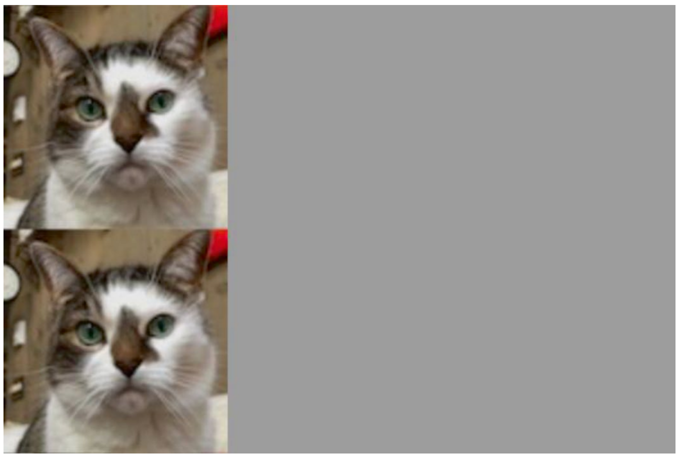


Figure 5-5. *The vertically repeated background image*

Moving the Background Image with the `background-position` Property

The position of the background image can be changed with the `background-position` property. Listing 5-6 shows an example of using `background-position` to place the background image in the center of the element.

Listing 5-6. Using the `background-position` property

```
<style>
    .cat {
        background-image: url('./small-cat.jpg');
        background-color: #999999;
        background-repeat: no-repeat;
        background-position: center;
        height: 200px;
        width: 300px;
    }
</style>

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

Figure 5-6 shows the centered background image.

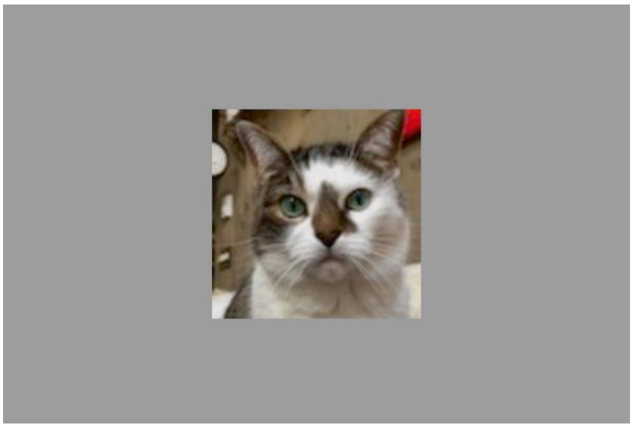


Figure 5-6. *The background image is centered in the element*

A single value can be given such as `top`, `bottom`, `left`, `right`, or `center`, a length such as `50px`, or a percentage. Two values can also be given. In this case, the first value is the position along the X-axis, and the second is the position along the Y-axis. This is demonstrated in Listing 5-7.

Listing 5-7. Specifying two values for `background-position`

```
<style>
    .cat {
        background-image: url('./small-cat.jpg');
        background-color: #999999;
        background-repeat: no-repeat;
        background-position: 50px center;
        height: 200px;
        width: 300px;
    }
</style>

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

The result is shown in Figure 5-7.

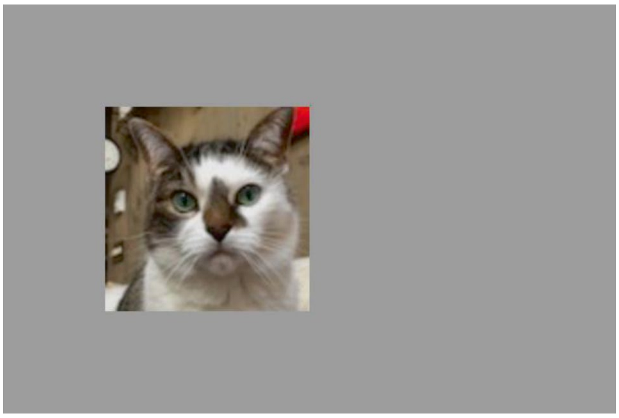


Figure 5-7. *The background image in the specified position*

Customizing the Size with the `background-size` Property

By default, the background image will keep its original size. This may not always be ideal and can be changed with the `background-size` property. In the following examples, we want to use the image from Figure 5-8 as a background image.

Figure 5-8. *Photo by Kalen Emsley on Unsplash*

First, we'll set a height and background image on the element in Listing 5-8.

Listing 5-8. Adding the background image

```
<style>
    .mountains {
        background-image: url('./mountains.jpg');
        height: 300px;
    }
</style>

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

Figure 5-9 shows the result. By default, we're seeing the top-left portion of the image.

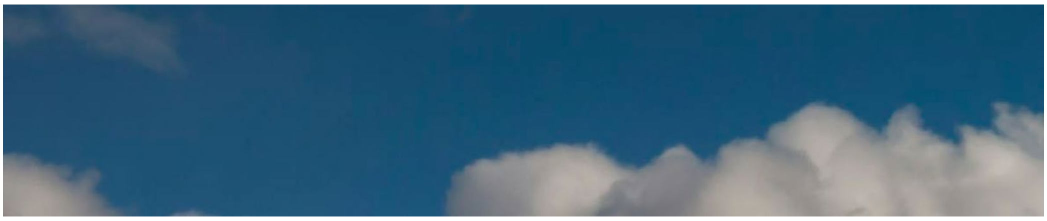


Figure 5-9. *The top-left portion of the background is applied*

The `background-size` property affects the size of the background image in its container. Here, we'll use the special value `cover`. This will resize the background image to make sure that the full image covers the element. If the aspect ratio of the image doesn't match that of the element, the image will be cropped.

In Listing 5-9, we'll apply this property to the background image to see the effect.

Listing 5-9. Applying a `background-size` of `cover`

```
<style>
    .mountains {
        background-image: url('./mountains.jpg');
        background-size: cover;
        height: 300px;
    }
</style>

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

The image is scaled so that it fits fully within the element. The element isn't tall enough to have the same aspect ratio, though, so the image is still cropped, as shown in Figure 5-10.

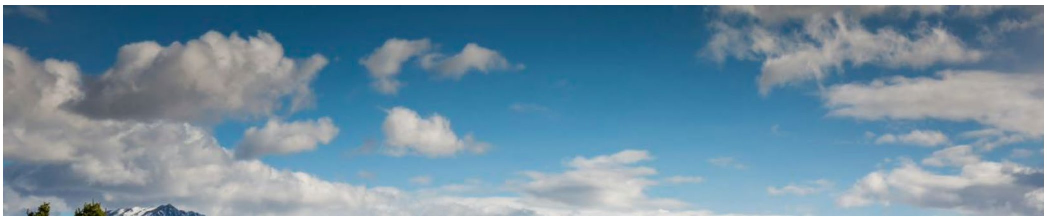


Figure 5-10. *The full width of the image is now contained within the element*

This looks better, but we still aren't seeing the main subject of the image – the mountains. Using a little trial and error, we can use the `background-position` property here to move the image, so that we get the portion of the image that we want as the background. This is done in Listing 5-10.

Listing 5-10. Adjusting the background position

```
<style>
    .mountains {
        background-image: url('./mountains.jpg');
        background-size: cover;
        background-position: 0 30%;
        height: 300px;
    }
</style>

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

As Figure 5-11 shows, now the mountains are visible in the element's background.

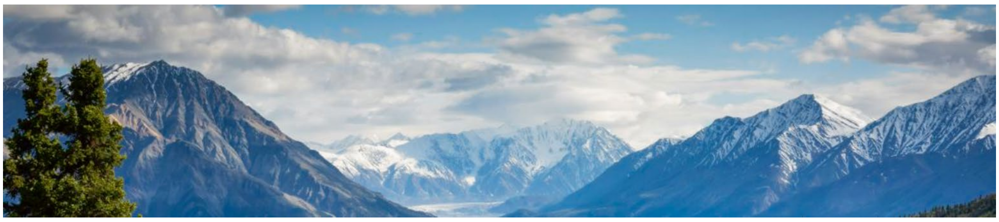


Figure 5-11. *The adjusted background position*

As mentioned earlier, `background-size: cover` will scale the background image but retain the aspect ratio. This is why you can't see all of the image. If you resize the window so the element becomes narrower, closer to the original image's aspect ratio, you'll see more of the image. This is shown in Figure 5-12.

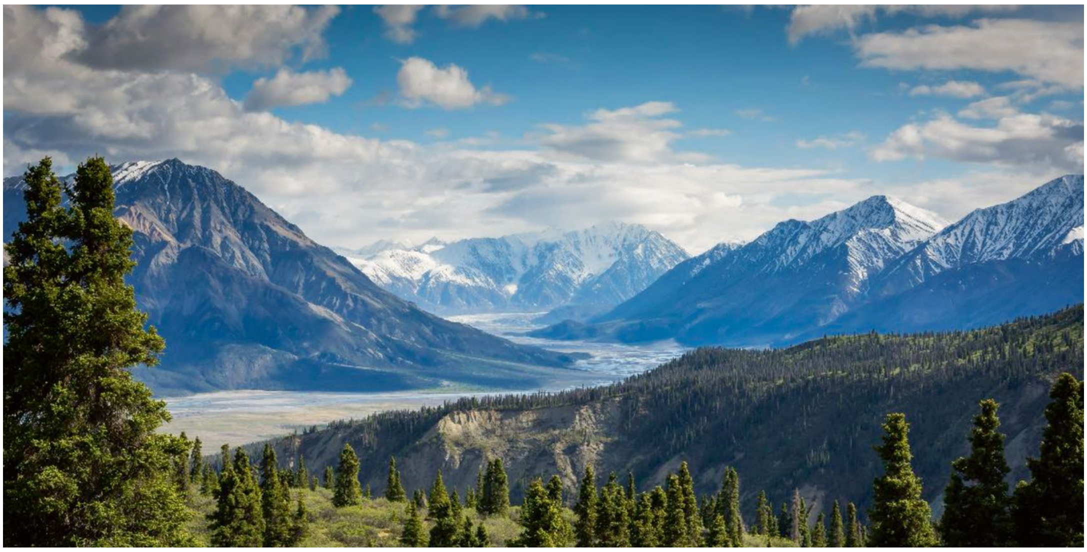


***Figure 5-12.** The image in a different aspect ratio*

Another value for the `background-size` property is `contain`. This resizes the image so it fits within the element. With the default `background-repeat` behavior, this causes the background image to be repeated, as shown in Figure 5-13.

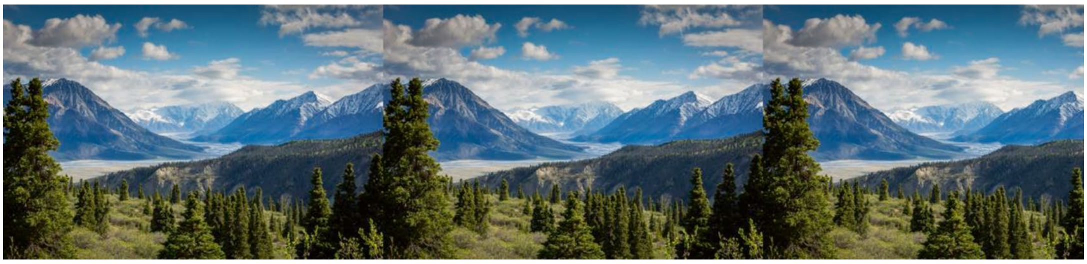


***Figure 5-13.** The background image fits and is repeated*

You can also give `background-size` a specific pixel value. This will cause the image to look "squished" if the aspect ratio is not preserved. Listing 5-11 shows an example of setting a specific size for the `background-size` property.

Listing 5-11. Using a specific background size

```
<style>
    .mountains {
        background-image: url('./mountains.jpg');
        background-size: 100px 200px;
        height: 300px;
    }
</style>

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

The aspect ratio is not preserved here, so the image looks “squished” and is tiled repeatedly, as shown in Figure 5-14.

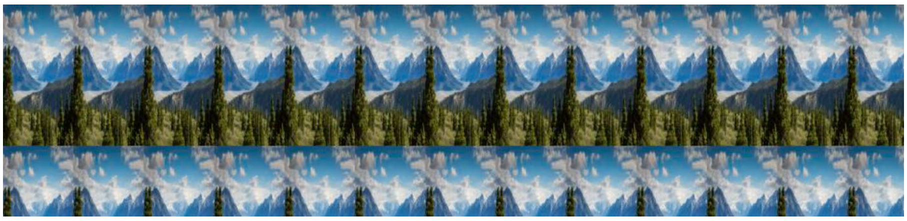


Figure 5-14. *The background image has a custom size*

The `background-clip` Property

The `background-clip` property is best explained with an example. Let’s return to the background image from Figure 5-11, where the background looks nice, and add some padding and a border. We’ll do this in Listing 5-12.

Listing 5-12. Adding padding and border to the element

```
<style>
    .mountains {
        background-image: url('./mountains.jpg');
        background-size: cover;
        background-position: 0 30%;
        height: 300px;
        border: 10px dashed red;
        padding: 32px;
    }
</style>

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

If you look closely at the element, you'll notice that the background image extends into the border area, and it is shown behind the border, as Figure 5-15 shows.

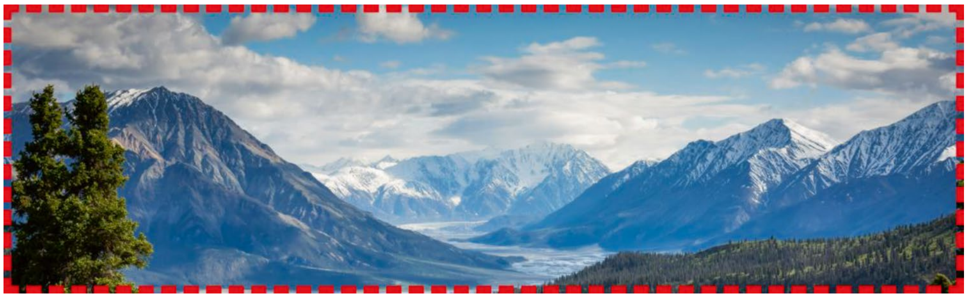


Figure 5-15. *The background extends underneath the border*

If this isn't what you want, you can change this behavior with the background-clip property. The default value, border-box, results in the behavior shown in Figure 5-15.

If you specify a value of padding-box, then the background image is clipped around the border area. This is shown in Figure 5-16.

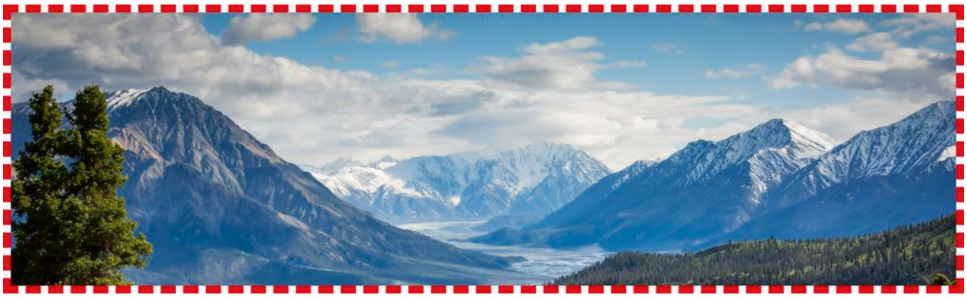


Figure 5-16. *The background does not extend underneath the border*

Finally, you can also specify a value of content-box. When you use this value, the background image is also clipped by the padding area and is only shown inside the element's content area, as shown in Figure 5-17.

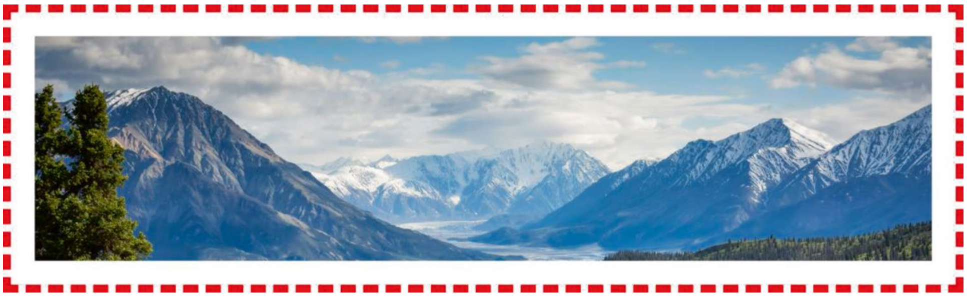


Figure 5-17. *The background image is only shown in the content area*

Using the Shorthand background Property

You can use the `background` property, a shorthand property, to set several of these properties in a single value. A background color can also be included. Listing 5-13 shows an example of using the `background` property.

Listing 5-13. Using the shorthand `background` property

```
<style>
    .mountains {
        background: url('./mountains.jpg') 0 30% / cover;
        height: 300px;
    }
</style>

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

This sets the background image URL, applies a `background-position` of `0 30%`, and a `background-size` of `cover`. The resulting element is shown in Figure 5-18.

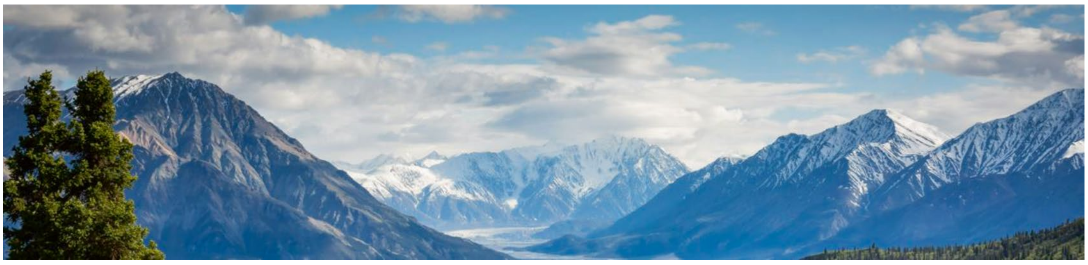


Figure 5-18. *Multiple background properties applied to the element*

Customizing Scroll Behavior with the `background-attachment` Property

By default, if an element contains scrollable content, the background image does not scroll with the content. It remains static as you scroll. However, it *does* scroll with the viewport. You can change this behavior with the `background-attachment` property.

If this is set to `fixed`, the background remains fixed relative to the viewport as well, so even as you scroll the whole page, the background remains static.

You can also set `background-attachment` to `local`. When this is set, the background image scrolls both with the viewport and with the element itself.

Background Gradients

In addition to solid colors and images, CSS also supports creating gradient backgrounds. There are three main types of gradients supported by modern browsers:

- *Linear gradients* go along a straight line. They can go left to right, top to bottom, or at any arbitrary angle.
- *Radial gradients* start at a central point and radiate outward.
- *Conic gradients* rotate the gradient around a central point.

There is not a CSS property for gradients. Instead, they are treated as background images. Gradients are specified with the `linear-gradient`, `radial-gradient`, and `conic-gradient` functions within the value for the `background-image` property.

Linear Gradients

A linear gradient gradually transitions between colors along a straight line. A gradient can have multiple "stops" or color transition points. Listing 5-14 shows a simple example of a linear gradient.

Listing 5-14. A simple linear gradient

```
<style>
    .gradient {
        background-image: linear-gradient(red, blue);
        height: 150px;
        width: 300px;
    }
</style>

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

The resulting gradient is shown in Figure 5-19.

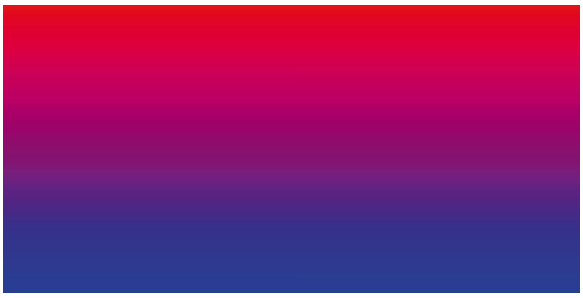


Figure 5-19. *The rendered gradient*

Adding Color Stops

You can add another color stop to a linear gradient by adding another color in the list, as shown in Listing 5-15.

Listing 5-15. A gradient with three color stops

```
<style>
    .gradient {
        background-image: linear-gradient(red, blue, green);
        height: 150px;
        width: 300px;
    }
</style>

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

The gradient now has a transition to green, as shown in Figure 5-20.

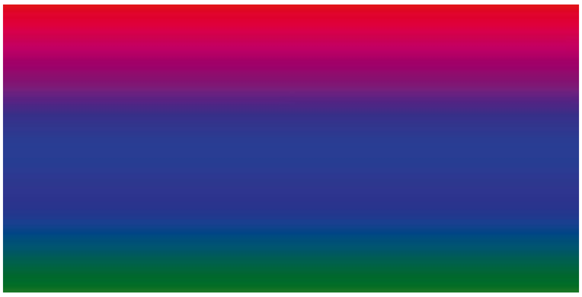


Figure 5-20. *The rendered three-stop gradient*

Using Transparency

You can use transparency in a gradient by specifying the color value `transparent` as gradient color stops, as shown in Listing 5-16.

Listing 5-16. Using transparency in a gradient

```
<style>
    .gradient {
        background-image: linear-gradient(transparent, blue, transparent);
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-21 shows the transparent gradient on a white background.

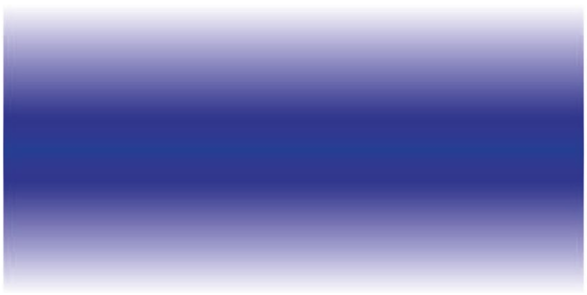


Figure 5-21. *The gradient with transparent color stops*

Changing the Gradient Direction

So far, all the gradients we've seen have been in the default direction – top to bottom. To change the direction of the gradient, add the argument `to <direction>` at the beginning of the linear-gradient function. This specifies the direction of the gradient. Listing 5-17 shows how we can have a gradient going from left to right.

Listing 5-17. A horizontal gradient

```
<style>
    .gradient {
        background-image: linear-gradient(
            to right,
            red,
            blue
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-22 shows the horizontal gradient, starting from red on the left and moving to the right to blue.

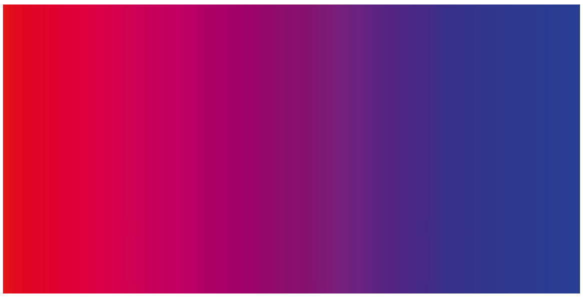


Figure 5-22. *The rendered horizontal gradient*

You can also specify an arbitrary angle for the linear gradient. Listing 5-18 shows an angle of 45 degrees to create a diagonal gradient.

Listing 5-18. Specifying an angle for the gradient

```
<style>
    .gradient {
        background-image: linear-gradient(
            45deg,
            red,
            blue
        );
        height: 150px;
        width: 300px;
    }
</style>

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

This results in a linear gradient that starts at the bottom-left corner and moves in a straight line at a 45-degree angle, as shown in Figure 5-23.

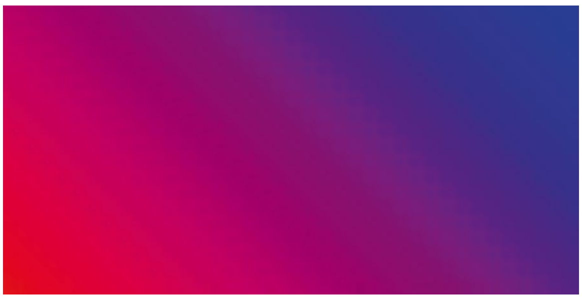


Figure 5-23. *A 45-degree gradient*

Customizing Color Stops

So far, the linear gradients we've seen have an equal distribution of colors. The color stops are spaced equally across the gradient. You can change the position along the gradient where the color stops are by specifying a percentage after the color. Listing 5-19 shows such a gradient.

Listing 5-19. Customizing gradient stops

```
<style>
    .gradient {
        background-image: linear-gradient(
            to right,
            red 0%,
            blue 25%
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-24 shows the resulting gradient. Notice how the gradient starts with a solid red color, then gradually transitions to blue at the 25% mark.

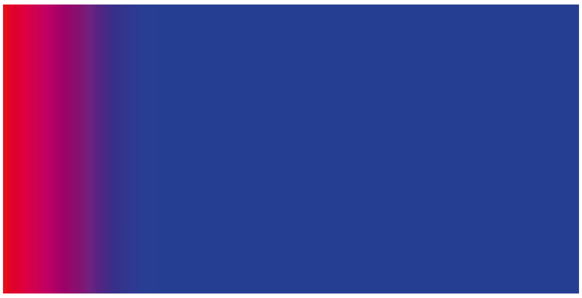


Figure 5-24. *The gradient with customized color stops*

Two adjacent stops with the same color create a region of solid color. Listing 5-20 has an example of creating stops like this.

Listing 5-20. A gradient with two adjacent stops of the same color

```
<style>
    .gradient {
        background-image: linear-gradient(
            to right,
            red 0%,
            green 25%,
            green 75%,
            blue 100%
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-25 shows the result. Note that between 25% and 75% along the gradient, the color is solid green.

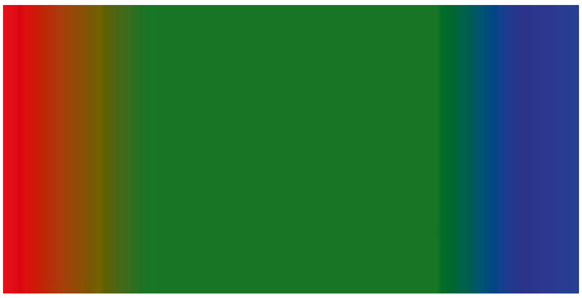


Figure 5-25. *A gradient with a solid region in the middle*

Similarly, if the first stop is after 0% or the last stop is before 100%, the remaining space before or after is also a solid color.

In addition to percentages, color stops can be specified with any valid length value in `px`, `em`, `rem`, or other units.

Radial Gradients

A radial gradient starts at a central point and radiates outward. Listing 5-21 has an example of a simple radial gradient.

Listing 5-21. A radial gradient

```
<style>
    .gradient {
        background-image: radial-gradient(red, blue);
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-26 shows the radial gradient. By default, the gradient's origin point is the center of the element.

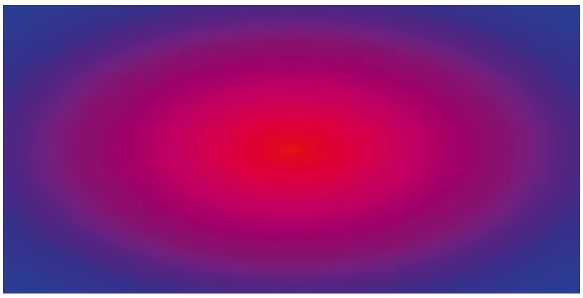


Figure 5-26. *A basic radial gradient*

Customizing the Shape and Position

The shape of a radial gradient can be an ellipse (the default) or a circle. You can define the gradient shape by specifying the name of the shape (`ellipse` or `circle`) and a position such as `top`, `right`, `left`, `center`, or specific percentages or values. The shape is defined as `<shape> at <position>`. The position can be omitted, in which case it defaults to the center of the image.

Listing 5-22 has an example of using a circle as the gradient shape.

Listing 5-22. Specifying a circle for the radial gradient shape

```
<style>
    .gradient {
        background-image: radial-gradient(
            circle,
            red,
            blue
        );
        height: 150px;
        width: 300px;
    }
</style>

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

The circular gradient is shown in Figure 5-27.

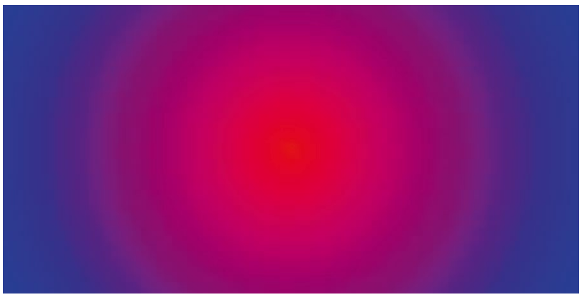


Figure 5-27. *The circular gradient*

You can move the circle to the left by specifying a position of 25%, as shown in Listing 5-23. The percentage represents the distance from the left side of the element.

Listing 5-23. Specifying the position of the gradient shape

```
<style>
    .gradient {
        background-image: radial-gradient(
            circle at 25%,
            red,
            blue
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-28 shows the new gradient position, 25% from the left side of the element.

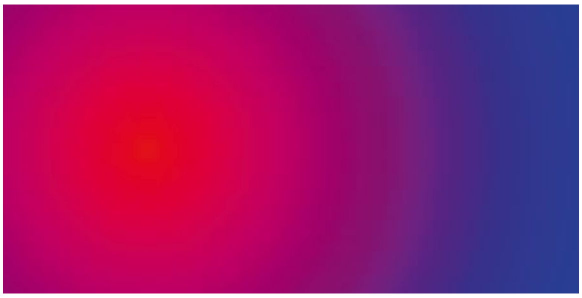


Figure 5-28. *The circle gradient at 25% from the left*

You can also specify two values for the position, in which case the first value is for the vertical position, and the second value is for the horizontal position. Listing 5-24 places the circular gradient at the top-left corner of the element.

Listing 5-24. Specifying two values for the gradient position

```
<style>
    .gradient {
        background-image: radial-gradient(
            circle at top left,
            red,
            blue
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-29 shows the result.

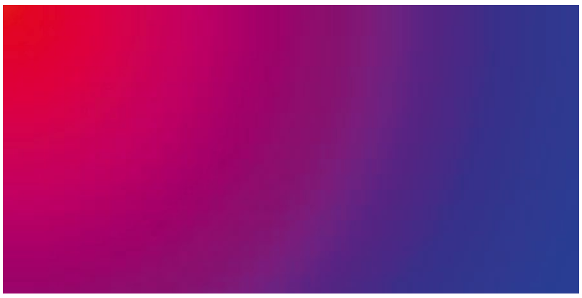


Figure 5-29. *The circular gradient in the top-left corner*

Customizing the Size

The size of the gradient can be further influenced by providing modifiers to the shape that define where the gradient should end. The options that can be set are

- `closest-side`: The gradient ends at the side closest to the center of the gradient. For a wide rectangle, this would be the top or bottom.
- `farthest-side`: The gradient ends at the side farthest from the center of the gradient. For a wide rectangle, this would be the left or right.
- `closest-corner`: The gradient ends at the closest corner to its center.
- `farthest-corner`: The gradient ends at the farthest corner from its center. This is the default.

Listing 5-25 shows an example of using the `closest-corner` modifier to make the gradient stop once it reaches the closest corner.

Listing 5-25. Using the `closest-corner` modifier

```
<style>
    .gradient {
        background-image: radial-gradient(
            circle closest-corner at 80% top,
            red,
            blue
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-30 shows the result. The gradient's center is along the top of the element, 80% of the way across. It begins radiating outward and stops once it hits the closest corner, which here is the top-right corner.

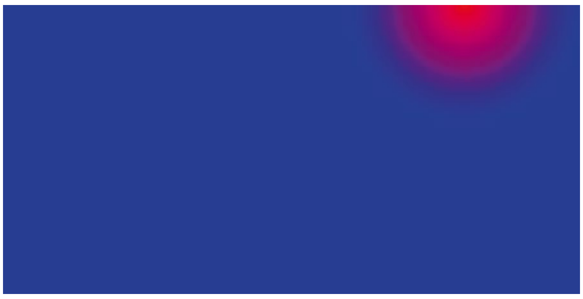


Figure 5-30. *The gradient using the `closest-corner` modifier*

Customizing Color Stops

Like linear gradients, radial gradients can also have multiple color stops. Listing 5-26 shows a radial gradient with three stops.

Listing 5-26. A radial gradient with multiple stops

```
<style>
    .gradient {
        background-image: radial-gradient(
            red,
            blue,
            green 75%
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-31 shows the gradient with red, blue, and green stops.

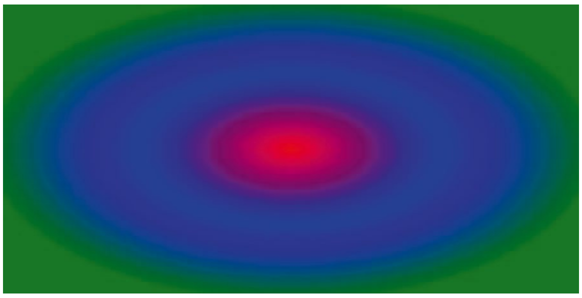


Figure 5-31. *The rendered gradient*

Multiple Gradients

You can even combine multiple radial gradients applied to an element if you use a transparent color as one of the stops, as demonstrated in Listing 5-27.

Listing 5-27. Multiple gradients

```
<style>
    .gradient {
        background-image: radial-gradient(
            ellipse at 25%,
            red,
            transparent
        ),
        radial-gradient(
            ellipse at 75%,
            blue,
            transparent
        );
        height: 150px;
        width: 300px;
    }
</style>

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

The result is two gradients, as shown in Figure 5-32.

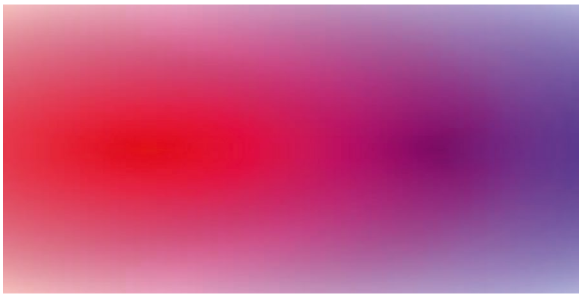


Figure 5-32. *The two gradients*

Conic Gradients

While radial gradients start at a central point and radiate outward, conic gradients form a circle around a point. By default, this is the center of the element. Listing 5-28 shows a simple example of a conic gradient.

Listing 5-28. A conic gradient

```
<style>
    .gradient {
        background-image: conic-gradient(
            red, orange, blue, green
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-33 shows the conic gradient. It starts at a 0-degree angle and rotates around the center point through the four color stops.

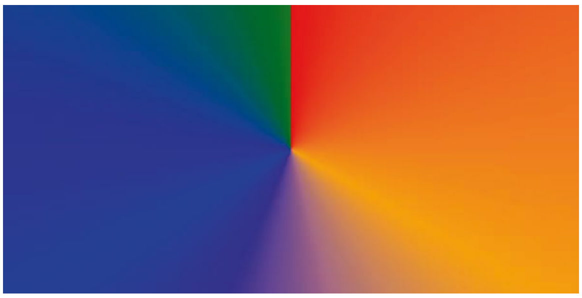


Figure 5-33. *A basic conic gradient*

Customizing the Angle

By default, the gradient starts at the 0-degree position. You can specify a different starting angle in the `conic-gradient` definition, as shown in Listing 5-29.

Listing 5-29. Specifying a different starting angle

```
<style>
    .gradient {
        background-image: conic-gradient(
            from 90deg,
            red, orange, blue, green
        );
        height: 150px;
        width: 300px;
    }
</style>

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

As Figure 5-34 shows, the gradient now starts at the 90-degree position.

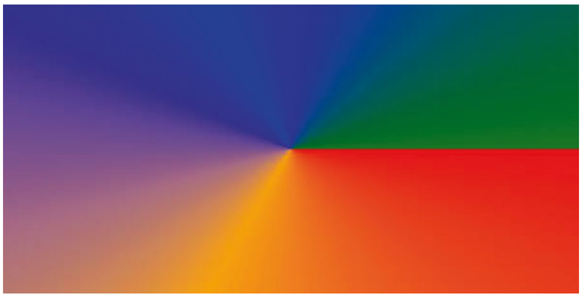


Figure 5-34. *The conic gradient starting at the 90-degree position*

Customizing Color Stops

The default behavior is to evenly transition between all the color stops. You can customize this behavior by specifying angles at which you want color stops to be placed. Listing 5-30 customizes the angles of the color stops so that they are all close together.

Listing 5-30. Customizing conic gradient stops

```
<style>
    .gradient {
        background-image: conic-gradient(
            red 90deg,
            orange 100deg,
            blue 120deg,
            green 130deg
        );
        height: 150px;
        width: 300px;
    }
</style>

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

Figure 5-35 shows the resulting conic gradient. Note that since the last stop occurs at 130 degrees, instead of 360 degrees, the rest of the gradient is solid green.

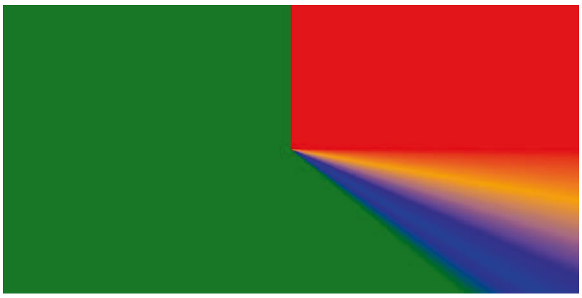


Figure 5-35. *The customized conic gradient*

You can also specify start and end angles for each stop to make hard color transitions, as shown in Listing 5-31.

Listing 5-31. Specifying start and end angles for each stop

```
<style>
    .gradient {
        background-image: conic-gradient(
            red 90deg 100deg,
            orange 100deg 120deg,
            blue 120deg 130deg,
            green 130deg
        );
        height: 150px;
        width: 300px;
    }
</style>

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

The gradient stops are hard transitions instead of gradual ones, as Figure 5-36 shows.

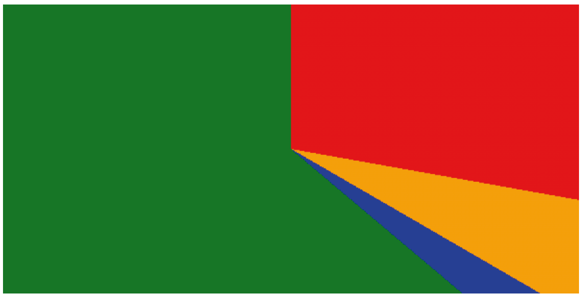


Figure 5-36. *Using hard color stops*

Combining Backgrounds

Gradients and background images can be combined to achieve light and shadow effects. Listing 5-32 has an example of applying a lighting effect with a gradient that goes from white to transparent.

Listing 5-32. Combining background images and gradients

```
<style>
    .mountains {
        background-image: radial-gradient(
            ellipse at top left,
            white 25%,
            transparent
        ),
        url('./mountains.jpg');
        background-size: cover;
        height: 300px;
    }
</style>

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

The result is a white light effect over the mountain image, as shown in Figure 5-37.

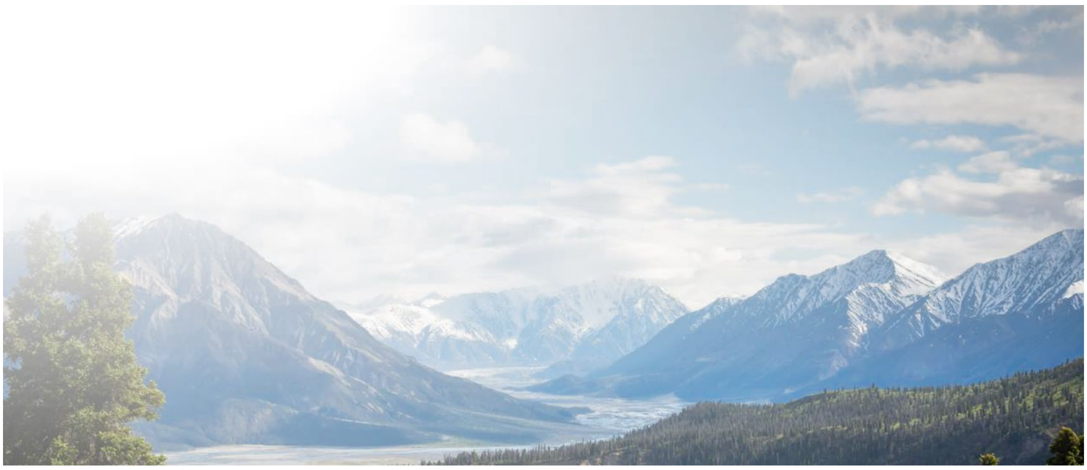


***Figure 5-37.** The gradient combined with a background image*

Similarly, a shadow effect could be achieved by using a dark color such as gray or black instead of white.

Summary

- An element's background can be a solid color, an image, a gradient, or a combination of the three.
- The display of a background image can be customized with the `background-repeat`, `background-position`, `background-size`, `background-clip`, and `background-attachment` properties.
- Gradients can be linear, radial, or conic.

CHAPTER 6

Text Styling

Text styling is one of the most common applications of CSS.

Basic Text Styling

In this section, we'll explore a few CSS properties that control basic text styling.

font-family

The `font-family` property sets the font to use for the element's text. This font is inherited by descendant elements.

`font-family` can be specified as a single value – the name of the font to use. More commonly, a comma-separated list of fonts is given. The browser will try each font, starting with the first, until a match is found.

Generally, the list starts specific and gets more general. The last font family in the list is typically a generic one like `monospace` or `sans-serif`, where the browser will use a fallback font that approximates the desired appearance. Font names containing spaces should be enclosed in quotes, as shown in Listing 6-1.

Listing 6-1. Specifying multiple font names

```
.hello {
    font-family:
        Georgia,
        'Times New Roman',
        serif;
}
```

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_6

In this example, the browser will try Georgia first. If Georgia is unavailable, it will try Times New Roman. Lastly, if that is not available, it will fall back to a built-in generic serif font. Custom web fonts can also be used. We will discuss that a little later.

Font Size

An element inherits its parent's font size by default. This behavior can be overridden by using the `font-size` property, which sets the font size for the element. Recall that the document has a base font size – usually `16px`.

The value of `font-size` not only controls the size of the text, but it also determines base sizing for anything specified in `em` or other relative units. The `em` unit is not just for text. Borders, padding, and even width and height can all be specified in `em` units.

In addition, there are several predefined `font-size` values, ranging from `xx-small` to `xxx-large`. A relative font size can also be specified, with a value of `smaller` or `larger`. A `font-size` can also be specified as a percentage of its parent's size.

As mentioned in Chapter 3, a `font-size` specified in `em` units has a compounding effect if its children also use `em` units. Listing 6-2 shows an example of this effect.

Listing 6-2. Using em units on the parent and child

```
<style>
  .parent {
    font-size: 1.5em;
  }

  .child {
    font-size: 1.5em;
  }
</style>

<div class="parent">
  I'm the parent
  <div class="child">
    I'm the child
  </div>
</div>
```

Figure 6-1 shows the resulting text.

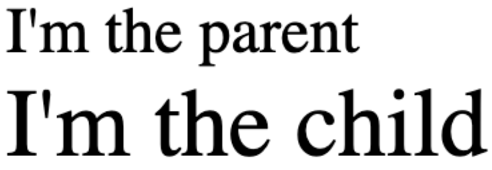


***Figure 6-1.** The parent and child have different font sizes*

Note that while the parent and child elements both have a `font-size` of `1.5em`, the child's text is larger. This is the compounding effect. The parent's font size is `1.5em`, or 1.5 times its parent's font size, which is the root element's font size of `16px`. This comes out to `24px`.

The child element's font size is also `1.5em`, 1.5 times of its parent, which we just calculated to be `24px`. `24px` * 1.5 = `36px`.

If you used `rem` units for the parent and child, they would have the same font size because `rem` is relative to the root element's font size.

Text Color

The `color` property controls the element's text color (and text decorations such as underlines). It also sets the current color. This is a special value called `currentColor` that resolves to the text color, which can be referenced from other properties. `currentColor` is also the default border color if one is not specified. Listing 6-3 has an example of `color` and `currentColor` in action.

Listing 6-3. The `color` property and `currentColor` value

```
<style>
    div {
        border: 3px solid currentColor;
    }

    .one {
        color: red;
    }
    .two {
        color: blue;
    }
</style>

<div class="one">One</div>
<div class="two">Two</div>
```

The result is shown in Figure 6-2.

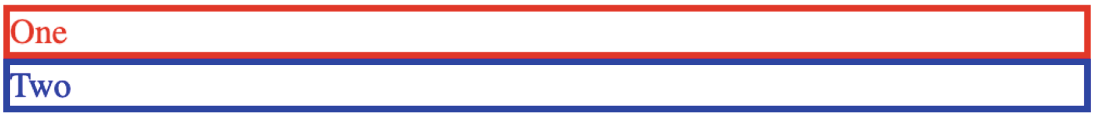


***Figure 6-2.** The two elements with different colors*

We give one element a color of `red` and the other a color of `blue`. This sets the `currentColor` value of each element. Then, we also have a rule that selects both `div` elements and uses the `currentColor` as the border color. The red `div` gets a red border, and the blue `div` gets a blue border.

Font Weight

The `font-weight` property defines how bold the text appears. This can be a simple value like `normal` or `bold`. It can also take numeric values: `100`, `200`, `300`, `400`, `500`, `600`, `700`, `800`, and `900`. The higher the number, the bolder the font is.

The `normal` value is equivalent to a weight of 400, and the `bold` value is equivalent to a weight of 700. Depending on the font used, not all weights may be available.

`font-weight` can also be specified as the values `lighter` or `bolder`. These values are relative to the weight of the element's parent.

Font Style

The `font-style` property can be used to make text italic. It has three supported values: `normal`, `italic`, and `oblique`. Italic and oblique are similar, but slightly different. Italic is typically an angled font face, sometimes with a completely different design than the normal version. On the other hand, oblique is typically just the normal version but slanted.

Not all fonts include both an italic and oblique version. If this is the case, then the `italic` and `oblique` styles look the same.

Underlining with `text-decoration`

The text-decoration property can be used to add decorative lines to text. These can be underlines, strikethrough lines, and even wavy or dotted lines. The basic usage of `text-decoration` takes a simple value – `none`, `underline`, or `line-through` – as shown in Listing 6-4.

Listing 6-4. Demonstrating the basic usage of the `text-decoration` property

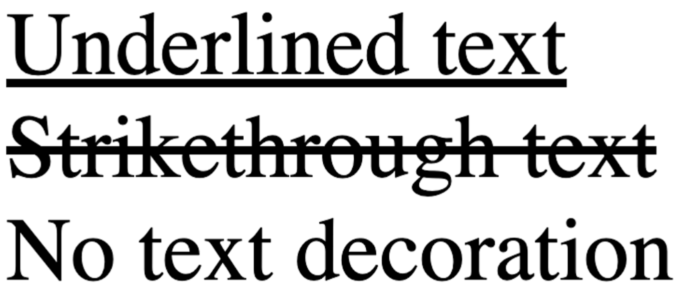
```
<style>
    .underline {
        text-decoration: underline;
    }

    .strikethrough {
        text-decoration: line-through;
    }

    .none {
        text-decoration: none;
    }
</style>

<div class="underline">Underlined text</div>
<div class="strikethrough">Strikethrough text</div>
<div class="none">No text decoration</div>
```

The resulting text style is shown in Figure 6-3.


Underlined text

~~Strikethrough text~~

No text decoration

Figure 6-3. *The basic text decoration types*

The `text-decoration` property can also take a color and a style. The available styles are `solid`, `double`, `dotted`, `dashed`, and `wavy`. These styles are shown in Listing 6-5.

Listing 6-5. Advanced text decoration styles

```
<style>
    .solid {
        text-decoration: underline solid blue;
    }

    .double {
        text-decoration: underline double green;
    }

    .dotted {
        text-decoration: underline dotted;
    }

    .dashed {
        text-decoration: underline dashed purple;
    }

    .wavy {
        text-decoration: underline wavy red;
    }
</style>
```

```
<div class="solid">Solid blue underline</div>
<div class="double">Double green underline</div>
<div class="dotted">Dotted black underline</div>
<div class="dashed">Dashed purple underline</div>
<div class="wavy">Wavy red underline</div>
```

These text decoration styles are shown in Figure 6-4.

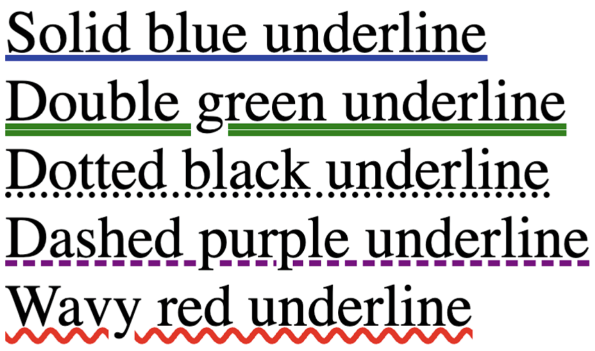


Figure 6-4. *The rendered advanced text decoration styles*

Some elements, such as links, have an underline by default. This can be removed by setting `text-decoration` to none.

You can also adjust the position of the text decoration by using the `text-underline-offset` property. This takes any valid CSS size expression and is shown in Listing 6-6.

Listing 6-6. Using the `text-underline-offset` property

```
<style>
    .underline {
        text-decoration: underline;
        text-underline-offset: 0.5em;
    }
</style>

<div class="underline">Underlined text</div>
```

The underline is moved down by `0.5em`, as shown in Figure 6-5.

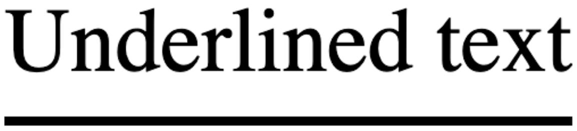


Figure 6-5. *The underline is moved by the `text-underline-offset` property.*

Other Text Effects

Transforming to Uppercase

The `text-transform` property can be used to transform the text to all uppercase or to capitalize just the first letter of every word. This property is shown in Listing 6-7.

Listing 6-7. Example usage of the `text-transform` property

```
<style>
    .uppercase {
        text-transform: uppercase;
    }

    .capitalize {
        text-transform: capitalize;
    }
</style>

<div class="uppercase">hello world</div>
<div class="capitalize">hello world</div>
```

The resulting text transformations are shown in Figure 6-6. Using `text-transform: uppercase` turns every letter to uppercase, whereas `text-transform: capitalize` only turns the first letter of each word to uppercase.

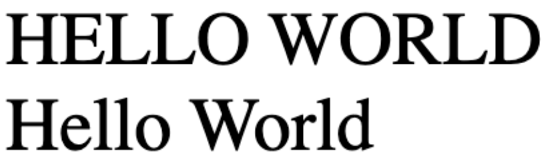


Figure 6-6. *The resulting text transformations*

The `text-transform` property also supports some other values, such as `none` (which doesn't change any of the letters) or `lowercase` (which makes everything lowercase).

Letter Spacing

The `letter-spacing` property is used to adjust the space between adjacent letters in a word. The specified value is not the spacing itself, but rather the amount of spacing to add to the normal spacing between the letters. Any valid CSS size value can be used, along with the keyword `normal`.

Listing 6-8 shows an example of using the `letter-spacing` property.

Listing 6-8. Using the `letter-spacing` property

```
<style>
    .hello {
        letter-spacing: 8px;
    }
</style>

<div class="hello">Hello world!</div>
```

This adds 8 pixels of extra spacing between characters, as shown in Figure 6-7.

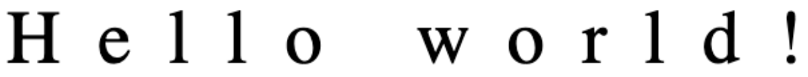


Figure 6-7. *The text with extra letter spacing*

Font Variant

The `font-variant` property can be set to `small-caps` for an interesting effect. All lowercase letters are transformed into smaller-sized capital letters. Listing 6-9 shows the usage of this property.

Listing 6-9. Using the `font-variant` property

```
<style>
    .hello {
        font-variant: small-caps;
    }
</style>

<div class="hello">Hello world!</div>
```

The lowercase letters are turned into small, capitalized letters, as shown in Figure 6-8.

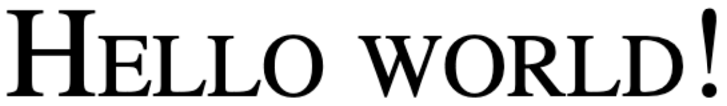


Figure 6-8. *The text with the small caps variant applied*

Text Layout

In addition to styling, there are also several useful properties that affect the text layout.

Text Indent

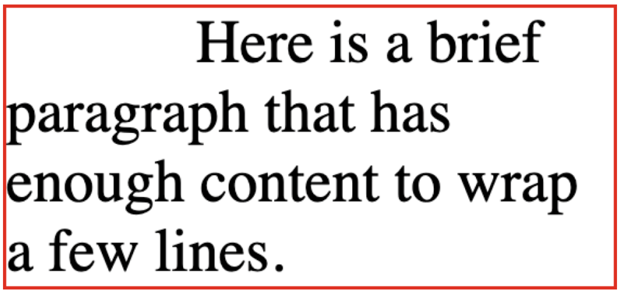
The `text-indent` property is used to specify an indent on the first line of text in a block element. Listing 6-10 shows how this property is used.

Listing 6-10. Using the `text-indent` property

```
<style>
    .my-text {
        border: 1px solid red;
        text-indent: 50px;
        width: 10rem;
    }
</style>
```

```
<div class="my-text">
    Here is a brief paragraph that has
    enough content to wrap a few lines.
</div>
```

Figure 6-9 shows the indented text.



Here is a brief paragraph that has enough content to wrap a few lines.

Figure 6-9. *The indent of 50 pixels applied to the first line of the text*

Whitespace

The `white-space` property is used to specify how whitespace is handled inside an element that contains text. The default value is `normal`. With this value, sequential whitespace characters are collapsed. If the text content exceeds the width of its container, it will be wrapped to the next line.

You may have seen this behavior before when you have multiple consecutive spaces in your HTML, or line breaks, and they are ignored by the browser. This is demonstrated in Listing 6-11.

Listing 6-11. An example with extra whitespace

```
<style>
    .my-text {
        border: 1px solid red;
        width: 10rem;
    }
</style>
```

```
<div class="my-text">
    Here is       some text
        with
            whitespace.
</div>
```

The whitespace behavior is shown in Figure 6-10.

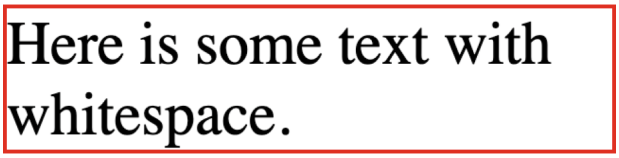


Figure 6-10. *The whitespace is ignored*

The extra spaces and line breaks are ignored, and the text only breaks to the next line when it automatically wraps.

We can make the browser respect the whitespace by setting `white-space` to `pre`, as shown in Listing 6-12.

Listing 6-12. Setting `white-space` to `pre`

```
<style>
    .my-text {
        border: 1px solid red;
        width: 10rem;
        white-space: pre;
    }
</style>

<div class="my-text">
    Here is       some text
        with
            whitespace.
</div>
```

As Figure 6-11 shows, the whitespace is now preserved in the rendered text.

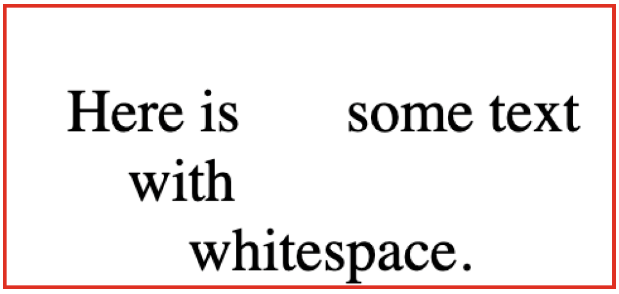


Figure 6-11. *The whitespace is preserved*

Notice how the whitespace is preserved in the rendered output now. You might also notice that there is an extra blank line at the top of the element. This represents the first line break after the opening `div` tag.

When `white-space` is set to `pre`, lines of text are not automatically wrapped. Some other accepted values for the `white-space` property are

- `normal`: The default behavior. Whitespace is collapsed, and text is automatically wrapped as needed.
- `nowrap`: Same as `normal`, except that lines of text do not wrap.
- `pre-wrap`: Same as `pre`, except that lines of text are also wrapped.
- `pre-line`: Same as `pre-wrap`, except that consecutive whitespace characters are collapsed. Line breaks are still preserved.

Truncating Text

Your design may require that text fit within its container without overflowing or wrapping. This can easily be accomplished by using the `white-space`, `overflow`, and `text-overflow` properties together.

First, `white-space` is set to `nowrap`. This ensures the text does not wrap but will instead cause the text to overflow the container. By setting `overflow` to `hidden`, you can hide the overflowing content. However, the text is abruptly cut off at the end of the container.

Finally, you can set `text-overflow` to `ellipsis` to truncate the text and add an ellipsis at the end. An example of this truncation technique is shown in Listing 6-13.

Listing 6-13. Truncating text

```
<style>
    .my-text {
        border: 1px solid red;
        overflow: hidden;
        text-overflow: ellipsis;
        white-space: nowrap;
        width: 10rem;
    }
</style>

<div class="my-text">
    Here is a really
    really long string.
</div>
```

The truncated text is shown in Figure 6-12.

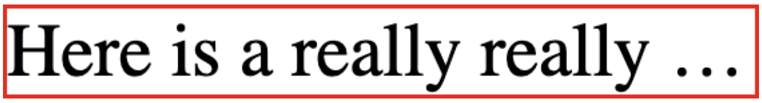


Figure 6-12. *The overflowing text is truncated with an ellipsis*

One thing to note with this technique is that it requires the width to be set. Otherwise, the text may not be truncated.

Horizontal Alignment

Horizontal alignment is controlled by the `text-align` property. This only has an effect on block elements with a width greater than that of their content. This property is demonstrated in Listing 6-14.

Listing 6-14. Controlling the text alignment

```
<style>
    .hello {
        width: 20rem;
        border: 1px solid red;
        margin: 8px;
    }

    .center {
        text-align: center;
    }

    .left {
        text-align: left;
    }

    .right {
        text-align: right;
    }
</style>

<div class="hello center">Hello world!</div>
<div class="hello left">Hello world!</div>
<div class="hello right">Hello world!</div>
```

The effect on alignment is shown in Figure 6-13.

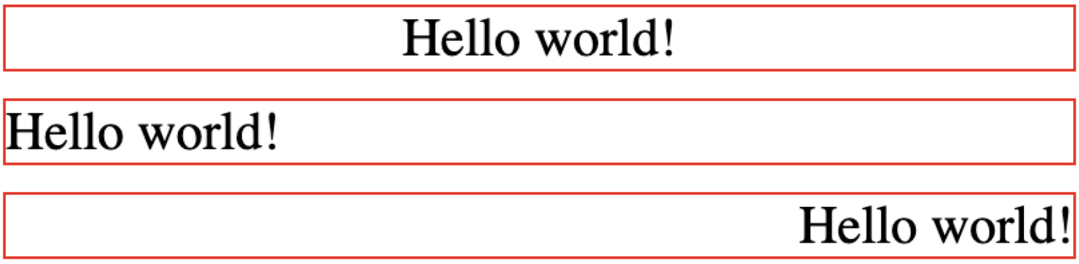


Figure 6-13. *Various text alignment options*

`text-align` doesn't just affect text. It sets the horizontal alignment of any inline element inside the containing element on which `text-align` is set.

Vertical Alignment

If a block element's height is taller than its content, by default the text will be aligned to the top of the container, as shown in Listing 6-15.

Listing 6-15. The default vertical alignment behavior

```
<style>
    .hello {
        height: 2em;
        width: 10em;
        border: 1px solid red;
    }
</style>

<div class="hello">Hello world!</div>
```

The default behavior is shown in Figure 6-14.

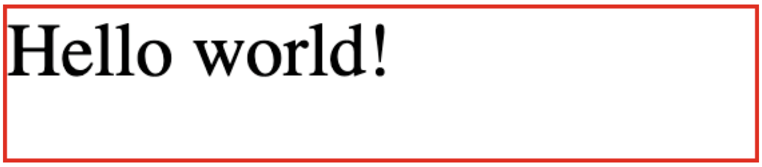


Figure 6-14. *The text defaults to top vertical alignment*

You might think the `vertical-align` property would help here, but setting `vertical-align: center` has no effect. This is because the `vertical-align` property only applies to inline elements, and a `div` is a block element.

This can be solved in a few ways, but one way is to use the `line-height` property. You can set the `line-height` equal to the element's height, as shown in Listing 6-16.

Listing 6-16. Vertically centering text using the `line-height` property

```
<style>
    .hello {
        height: 2em;
        line-height: 2em;
        width: 10em;
        border: 1px solid red;
    }
</style>

<div class="hello">Hello world!</div>
```

As Figure 6-15 shows, this centers the text vertically.

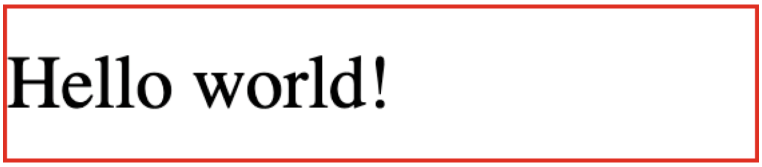


Figure 6-15. *The text is vertically centered*

How does the `vertical-align` property work, then? It controls how inline elements are aligned vertically with each other. Listing 6-17 has two `span` elements side by side with different heights inside a container element.

Listing 6-17. Vertically aligning two span elements

```
<style>
  .container {
    border: 2px dashed blue;
    width: 20rem;
    text-align: center;
  }

  .hello {
    border: 2px solid red;
    font-size: 2rem;
  }

  .world {
    border: 2px solid red;
    font-size: 4rem;
  }
</style>

<div class="container">
  <span class="hello">Hello</span>
  <span class="world">World!</span>
</div>
```

Figure 6-16 shows the vertical alignment of the two span elements.

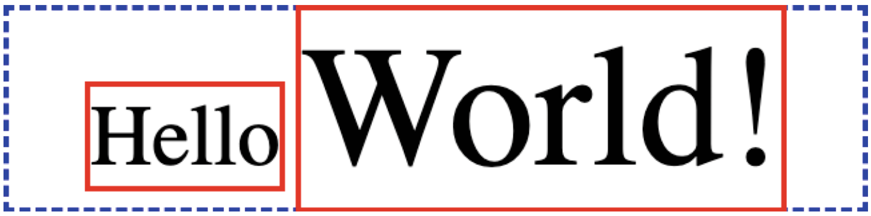


Figure 6-16. *The rendered aligned elements*

In Figure 6-16, you can see that the elements are aligned along their baselines. A baseline is an invisible line along which most letters sit. If we set `vertical-align` to `middle` on both elements, they become vertically aligned with each other. This is shown in Listing 6-18.

Listing 6-18. Setting `vertical-align` to `middle`

```
<style>
    .container {
      border: 2px dashed blue;
      width: 20rem;
      text-align: center;
    }

    .hello {
      border: 2px solid red;
      font-size: 2rem;
      vertical-align: middle;
    }

    .world {
      border: 2px solid red;
      font-size: 4rem;
      vertical-align: middle;
    }
  </style>
```

```
<div class="container">
  <span class="hello">Hello</span>
  <span class="world">World!</span>
</div>
```

As Figure 6-17 shows, the elements are now vertically aligned in the middle.

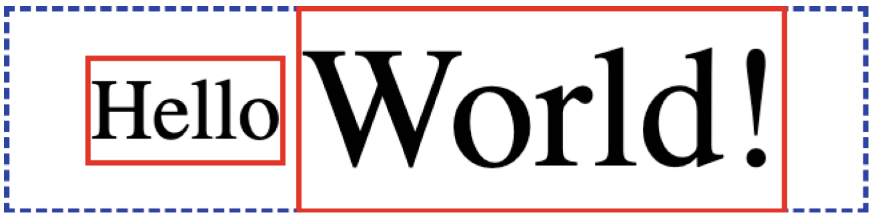


***Figure 6-17.** The elements are now vertically aligned.*

Using Web Fonts

If you don't want to use the web safe fonts (and who wants to see *another* website in Arial or Times New Roman?), you are in luck. Web fonts allow the CSS to link to a font file that the browser can download.

By using a web font, you can have a much more consistent look – plus, there are many beautiful web fonts out there that will enhance the look of your site or app.

There are several different supported web font formats:

- Web Open Font Format version 1 or 2 (WOFF/WOFF2)
- Embedded Open Type (EOT)
- TrueType Font (TTF)
- Scalable Vector Graphics (SVG)

Modern browsers support WOFF and WOFF2. The other font formats are for support with older browsers. A web font is typically packaged in several different formats, all of which can be referenced in the CSS.

Registering a Font with the `@font-face` Rule

A web font is registered using the `@font-face` at-rule. A `@font-face` rule declares a new font. The desired name of the font is given with the `font-family` property, and one or more source URLs are given with the `src` property. Each source URL is followed by a format declaration which tells the browser which font format to expect for that file.

Once you have declared the font in a `@font-face` rule, you can then use the name you gave it in any `font-family` property in a CSS rule.

The example in Listing 6-19 will load the `SomeWebFont` font in WOFF2 and WOFF formats and set it as the font for the whole document. You should still provide a list of fallback fonts in case the font is not supported by the user's browser or could not be loaded.

Listing 6-19. Using a web font

```
@font-face {
  font-family: "SomeWebFont";
  src: url("/some-font.woff2") format("woff2"),
    url("/some-font.woff") format("woff");
}

body {
  font-family: SomeWebFont, Arial, sans-serif;
}
```

Declaring Different Web Font Styles

Usually, a given web font file is only a single weight or style version of the font. This means there is one font file for the normal version and another for the bold version.

They both must be registered in a separate `@font-face` rule under the same `font-family`. The font weight and style are specified via the `font-weight` and `font-style` properties. This is shown in Listing 6-20.

Listing 6-20. Defining two weights of a web font

```
@font-face {
  font-family: "SomeWebFont";
  src: url("/some-font.woff2") format("woff2");
  font-weight: 400;
}

@font-face {
  font-family: "SomeWebFont";
  src: url("/some-font-bold.woff2") format("woff2");
  font-weight: 700;
}
```

A Word of Caution on Web Fonts

Web fonts are great, but don't go overboard. The more fonts that are loaded, the longer the page takes to load, and the worse the flash of unstyled text can be. You should make sure to use only the web fonts that you absolutely need.

Text Shadow

The `text-shadow` property allows you to add shadows to text. It works similarly to the `box-shadow` property from Chapter 4. A text shadow has X and Y offsets, an optional blur radius, and a color. Unlike `box-shadow`, text-shadow does not have a spread radius.

Here are some examples of text shadows. Listing 6-21 uses a shadow with an offset but no blur.

Listing 6-21. A text shadow example

```
<style>
  .container {
    font-size: 2rem;
    font-family: Arial, sans-serif;
    text-shadow: 2px 2px 0px red;
  }
</style>

<div class="container">Hello World!</div>
```

The resulting text shadow is shown in Figure 6-18.

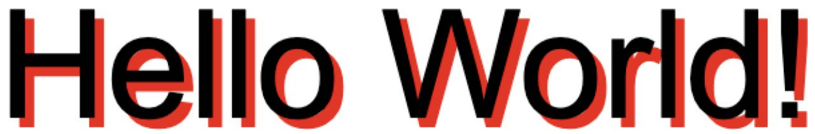


Figure 6-18. *A simple text shadow*

Listing 6-22 has another example of a text shadow, this time with no offset and a blur radius.

Listing 6-22. A second text shadow example

```
<style>
  .container {
    font-size: 2rem;
    font-family: Arial, sans-serif;
    text-shadow: 0px 0px 5px red;
  }
</style>

<div class="container">Hello World!</div>
```

The shadow, with its blur, can be seen in Figure 6-19.

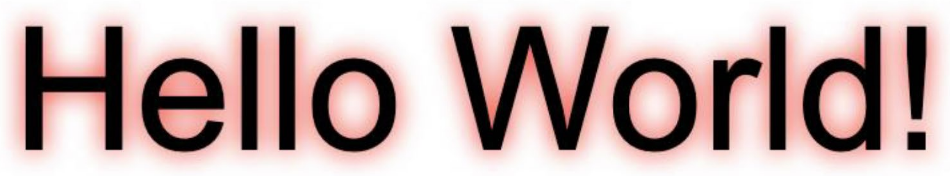


Figure 6-19. *A text shadow with a blur effect*

Applying a Gradient to Text

It’s also possible to apply a gradient effect to text. However, as of February 2025, this is only supported in WebKit-based browsers and requires a prefixed property name (`-webkit-text-fill-color`).

To apply this effect, you apply a gradient background to the element, as was demonstrated in Chapter 5. Then you can use the `background-clip` and `-webkit-text-fill-color` properties to apply the background gradient to the text. This is shown in Listing 6-23.

Listing 6-23. A text gradient effect

```
<style>
    .gradient {
        background: linear-gradient(red, blue);
        background-clip: text;
        -webkit-text-fill-color: transparent;
    }
</style>

<div class="gradient">Text Gradient</div>
```

If you do this in a WebKit-based browser (Chrome, Safari, Edge), you'll see a gradient applied to the text as shown in Figure 6-20.

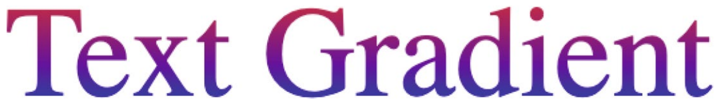


Figure 6-20. *The rendered text gradient, as shown in Chrome*

Summary

- You can use various CSS properties to control the font size, color, weight, and style.
- The `text-decoration` property can add underlines and strikethroughs.
- There are other text effect properties such as `text-transform`, `letter-spacing`, and `font-variant`.
- The `white-space` property controls how the browser renders whitespace. Whitespace can be ignored or respected.

- The `vertical-align` property controls how inline elements are vertically aligned with each other.
- Fonts can be downloaded by the browser and used with the `@font-face` rule.
- Text shadows can be applied with the `text-shadow` property.
- WebKit browsers can apply a text gradient by combining a background gradient with the `background-clip` and `-webkit-text-fill-color` properties.

CHAPTER 7

Layout and Positioning

So far, we've looked a lot at how to style with CSS. Let's switch gears now and look at how to lay out and position elements. We'll start by looking at padding and margin, which were briefly discussed as part of the CSS box model in Chapter 3.

Padding

The padding is the space between an element's content and its border, specified with the `padding` property. By default, most elements have no padding. An element's padding value is not inherited by its children. Listing 7-1 shows an element with no explicit padding applied.

Listing 7-1. An element with no padding

```
<style>
  .container {
    background: skyblue;
  }
</style>

<div class="container">
  Hello world!
</div>
```

As Figure 7-1 shows, there is no space between the content and the outer edge of the element.

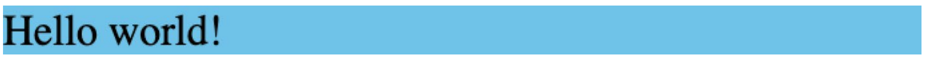


Figure 7-1. *The default style does not apply any padding*

Padding can be specified with any size unit or with a percentage. Listing 7-2 shows an example of adding padding to an element.

Listing 7-2. Adding padding to an element

```
<style>
  .container {
    background: skyblue;
    padding: 1rem;
  }
</style>

<div class="container">
  Hello world!
</div>
```

As Figure 7-2 shows, there is now `1rem` of padding around the element's content area.

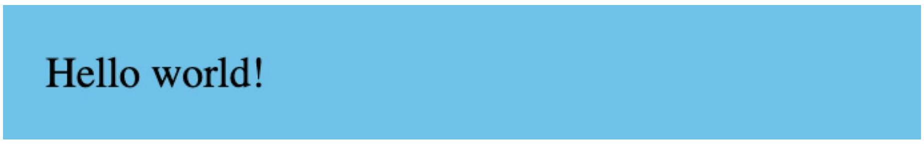


Figure 7-2. *The element with padding*

When padding is specified as a percentage, the value used is the given percentage of the containing block's width. Listing 7-3 has a simple example of this.

Listing 7-3. Setting a percentage value for padding

```
<style>
  .container {
    border: 1px solid red;
    width: 200px;
  }

  .inner {
    padding: 25%;
  }
</style>
```

```
<div class="container">
  <div class="inner">Hello world!</div>
</div>
```

The resulting element is shown in Figure 7-3.

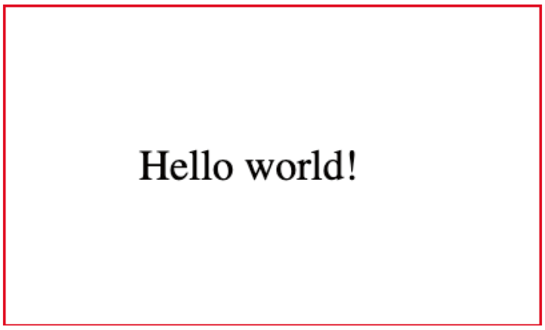
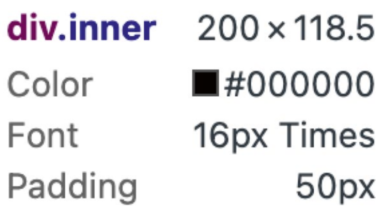


Figure 7-3. *The rendered result*

If you examine the inner element with the browser's developer tools, you'll see that the padding is `50px`, or 25% of the container's width of `200px`, as shown in Figure 7-4.



```
div.inner    200 × 118.5
Color        ■#000000
Font         16px Times
Padding           50px
```

Figure 7-4. *The dimensions of the inner element, shown in the Chrome developer tools*

Unlike margin, which we'll look at next, padding on adjacent elements does not collapse. The full padding amounts of each element are preserved. Consider the example shown in Listing 7-4.

Listing 7-4. Two adjacent elements with padding

```
<style>
    .inner1 {
        background: skyblue;
        padding: 50px;
    }
    .inner2 {
        background: lightgreen;
        padding: 50px;
    }
</style>

<div>
    <div class="inner1">Inner 1</div>
    <div class="inner2">Inner 2</div>
</div>
```

As Figure 7-5 shows, each element's padding of 50 pixels is preserved in all directions.

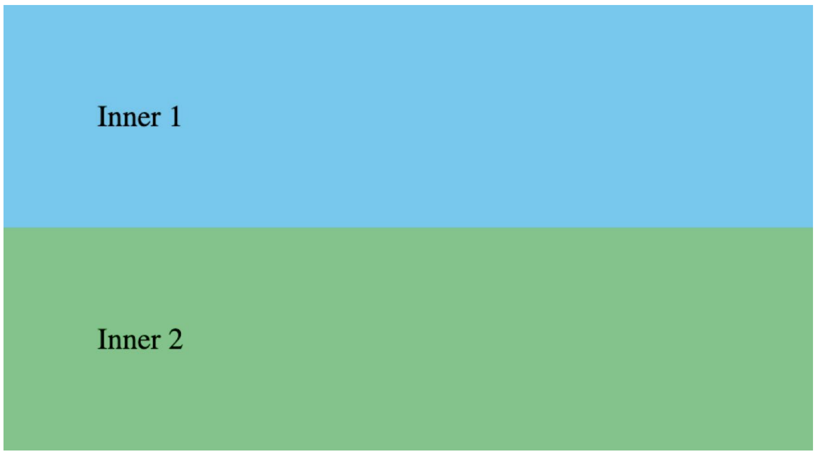


Figure 7-5. *Padding does not collapse*

Margin

An element's margin is the space between its border and other adjacent or containing elements. The value of the `margin` property can be a size value, a percentage, or the keyword `auto`.

By default, most elements have no margin. An example of this is shown in Listing 7-5.

Listing 7-5. Elements with no margin

```
<style>
  .container {
    border: 5px solid skyblue;
    width: 10rem;
  }

  .inner {
    border: 5px solid lightgreen;
  }
</style>

<div class="container">
  <div class="inner">Hello world!</div>
</div>
```

As Figure 7-6 shows, there is no space between the blue and green borders. This is because, by default, the elements have no margin.

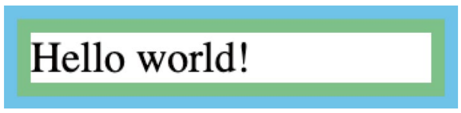


Figure 7-6. *The borders are adjacent, with no margin between them*

Listing 7-6 adds margin to the inner element.

Listing 7-6. Element with margin

```
<style>
  .container {
    border: 5px solid skyblue;
    width: 10rem;
  }

  .inner {
    border: 5px solid lightgreen;
    margin: 1rem;
  }
</style>
```

```
<div class="container">
  <div class="inner">Hello world!</div>
</div>
```

Now there is a space of `1rem` between the inner element's border and the outer element, due to margin, as shown in Figure 7-7.

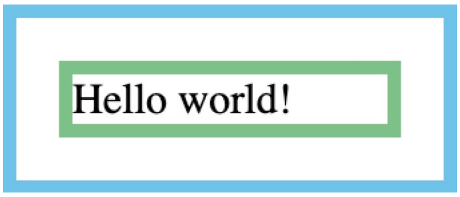


Figure 7-7. *Adding margin to the inner element*

Like with padding, using a percentage for the margin will set the margin to the given percentage of the containing block's width.

Centering an Element with `margin: auto`

The `margin` property also accepts the keyword `auto` as a valid value. When the horizontal (left and right) margin is set to auto in a block or inline-block element, the element is centered horizontally within its containing element.

The element takes the specified width, and the margin is automatically distributed evenly on the left and right. An example of this centering technique is shown in Listing 7-7.

Listing 7-7. Centering horizontally with margin: auto

```
<style>
    .container {
        background: skyblue;
        width: 20rem;
    }

    .inner {
        background: pink;
        margin: auto;
        width: 10rem;
    }
</style>

<div class="container">
    <div class="inner">
        Hello world!
    </div>
</div>
```

As Figure 7-8 shows, the inner element is centered horizontally due to the margin of `auto`.

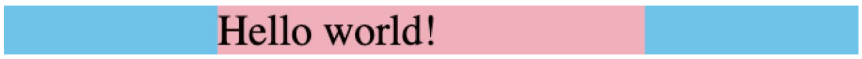


Figure 7-8. *The inner element is centered horizontally*

Using `margin: auto` works for horizontal centering, but it will not work for vertical centering. However, as we'll see later, there are several other ways to vertically center a block element.

Margin Collapse

When two elements with a vertical margin meet vertically, the two margins are collapsed into a single margin. The size of the collapsed margin depends on the size of the adjacent margins. If they are the same size, then the collapsed margin is the same size as the common margin. If they are different sizes, the collapsed margin takes the size of the larger margin. Margin collapse between adjacent elements applies to vertical margins only.

Margin collapse is demonstrated in Listing 7-8.

Listing 7-8. Demonstration of margin collapse

```
<style>
    .element1 {
        background: skyblue;
        margin-bottom: 1rem;
    }
    .element2 {
        background: lightgreen;
        margin-top: 1rem;
    }
</style>

<div>
    <div class="element1">Element 1</div>
    <div class="element2">Element 2</div>
</div>
```

The collapsed margin is shown in Figure 7-9. The top element has `1rem` of bottom margin, and the bottom element has `1rem` of top margin. But the space between the two is only `1rem`, because the margins have collapsed.

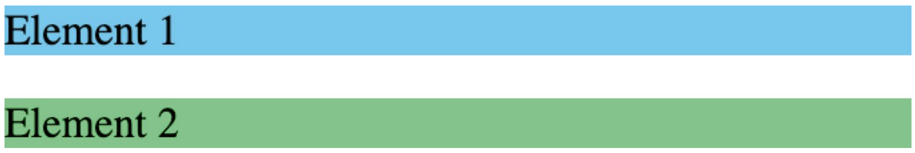


Figure 7-9. *The elements with the adjacent margins collapsed*

Another situation where the vertical margins collapse is when there is no border, padding, or other content between a parent element and its child, as shown in Listing 7-9.

Listing 7-9. Margin collapse between a parent and child element

```
<style>
    .container {
        background: skyblue;
        margin: 1rem;
    }

    .inner {
        background: lightgreen;
        margin: 1rem;
    }
</style>
```

```
<div class="container">
    <div class="inner">Inner</div>
</div>
```

The result is shown in Figure 7-10.

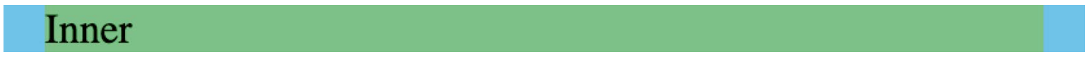


Figure 7-10. *The child element's margin is collapsed*

Because there is nothing separating the outer element from its content, the margin of the inner element is collapsed. In Listing 7-10, we add a border to the container element.

Listing 7-10. Adding a border to the outer element

```
<style>
    .container {
        background: skyblue;
        border: 1px solid red;
        margin: 1rem;
    }

    .inner {
        background: lightgreen;
        margin: 1rem;
    }
</style>

<div class="container">
    <div class="inner">Inner</div>
</div>
```

As Figure 7-11 shows, due to this border, the child element's margin no longer collapses and is shown between the inner and outer element.

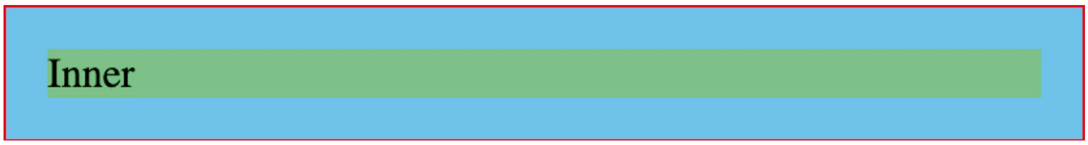


Figure 7-11. *The child element's margin is now visible*

Positioning Elements

The CSS `position` property determines how an element is positioned within the document. The `top`, `right`, `bottom`, and `left` properties are used with the `position` property to determine an element's final position. By default, elements use `static` positioning.

If an element's position property is set to any value other than `static`, it is considered a *positioned element*. This has important implications about the positioning of descendant elements.

Static Positioning

A statically positioned element is laid out in the normal flow of the document. If no value is given for the `position` property, or if it is set to `static`, the element will be statically positioned. When an element uses static positioning, the `top`, `right`, `bottom`, and `left` properties have no effect.

Relative Positioning

An element uses relative positioning if the `position` property is set to `relative`. The element is laid out in the normal flow of the document, as with static positioning, but is then moved according to the values of the `top`, `right`, `bottom`, and `left` properties.

When an element is moved from its default position using relative positioning, it doesn't affect the position of other elements in the document. To illustrate this, first consider the example in Listing 7-11 which defines three boxes.

Listing 7-11. An example layout containing three boxes

```
<style>
  .block {
    background-color: red;
    height: 64px;
    width: 64px;
    margin: 8px;
  }
  .green {
    background-color: green;
  }

  .blue {
    background-color: blue;
  }
</style>

<div class="block"></div>
<div class="block blue"></div>
<div class="block green"></div>
```

Figure 7-12 shows the rendered layout.

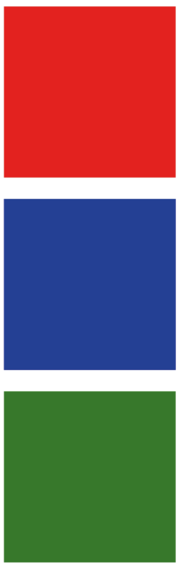


Figure 7-12. *The example layout*

The three boxes all use static positioning, so they appear at their default position within the normal document flow. Listing 7-12 changes the blue box to use relative positioning.

Listing 7-12. Applying position: relative

```
<style>
  .block {
    background-color: red;
    height: 64px;
    width: 64px;
    margin: 8px;
  }

  .green {
    background-color: green;
  }

  .blue {
    background-color: blue;
    position: relative;
    top: 32px;
    left: 32px;
  }
</style>

<div class="block"></div>
<div class="block blue"></div>
<div class="block green"></div>
```

The result is shown in Figure 7-13. The blue box has moved, but the other boxes remain in their original positions. As a result, the blue box partially overlaps the green box.

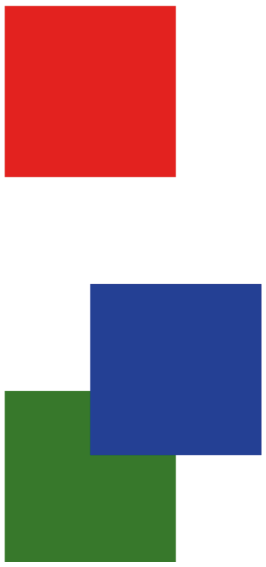


Figure 7-13. *The blue box is offset from its original position*

When a vertical or horizontal offset is given, the element is moved in the opposite direction. That is, `top` moves the element down, `left` moves the element to the right, `right` moves the element to the left, and `bottom` moves the element up.

What happens if you specify conflicting offsets? For example, an element can't be 10 pixels below its top position and 10 pixels above its bottom position and have the correct size. Generally, if both `top` and `bottom` are specified, the `top` value is used, and the `bottom` value is ignored. Similarly, if both `left` and `right` are specified, `left` wins if the text direction is left to right, and `right` wins if the text direction is right to left.

The `top`, `right`, `bottom`, and `left` properties can also have negative values. For example, specifying a `top` of `-32px` will move the element up by 32 pixels.

Absolute Positioning

An absolutely positioned element can also have `top`, `right`, `bottom`, and `left` offsets that affect its position. However, the layout is affected differently than with relative positioning.

To absolutely position an element, set its `position` to `absolute`. Absolute positioning removes the element from the normal document flow, and the element “floats” above the rest of the document. Unlike with relative positioning, the position of the other elements will be adjusted as if the absolutely positioned element is not there.

Consider the three-box layout from the previous example. Let's apply `position: absolute` to the middle blue box, as shown in Listing 7-13.

Listing 7-13. Applying position: absolute

```
<style>
  .block {
    background-color: red;
    height: 64px;
    width: 64px;
    margin: 8px;
  }

  .green {
    background-color: green;
  }

  .blue {
    background-color: blue;
    position: absolute;
    top: 32px;
    left: 32px;
  }
</style>

<div class="block"></div>
<div class="block blue"></div>
<div class="block green"></div>
```

Figure 7-14 shows the result. As before, the blue box is moved, but in this case the gap between the red and green boxes is gone.

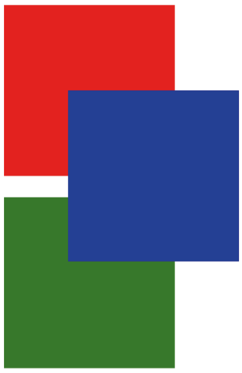


Figure 7-14. *The "hole" in the layout from the blue box is gone when position: absolute is applied*

There is also a difference in the interpretation of the `top`, `right`, `bottom`, and `left` offsets. While a relatively positioned element's offsets are relative to the element's original position in the document, an absolute positioned element's offsets are relative to the closest ancestor positioned element.

This is not necessarily the element's direct parent. An element is considered *positioned* if it has a `position` property set to something other than `static`. Listing 7-14 has an example layout that will have absolute positioning applied to it.

Listing 7-14. A layout with three boxes, one inside the next

```
<style>
  .outer {
    background-color: red;
    height: 120px;
    width: 120px;
  }

  .inner {
    background-color: blue;
    height: 80px;
    width: 80px;
  }
  .core {
    background-color: green;
    height: 40px;
    width: 40px;
  }
</style>

<div class="outer">
  <div class="inner">
    <div class="core"></div>
  </div>
</div>
```

This layout is shown in Figure 7-15.

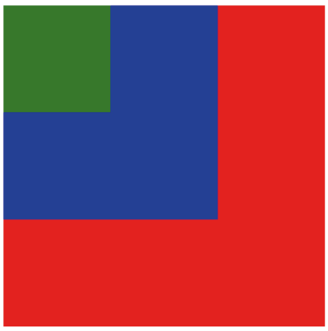


Figure 7-15. *The example layout before absolute positioning is applied*

These elements are all statically positioned. Suppose we want to move the green box all the way to the right edge of the outer red box. Let's try setting its `position` to `absolute` and its `right` offset to `0`. This is done in Listing 7-15.

Listing 7-15. Applying position: absolute

```
<style>
  .outer {
    background-color: red;
    height: 120px;
    width: 120px;
  }
  .inner {
    background-color: blue;
    height: 80px;
    width: 80px;
  }

  .core {
    background-color: green;
    height: 40px;
    width: 40px;
    position: absolute;
    right: 0;
  }
</style>

<div class="outer">
  <div class="inner">
    <div class="core"></div>
  </div>
</div>
```

The result, shown in Figure 7-16, may not be what you expect.

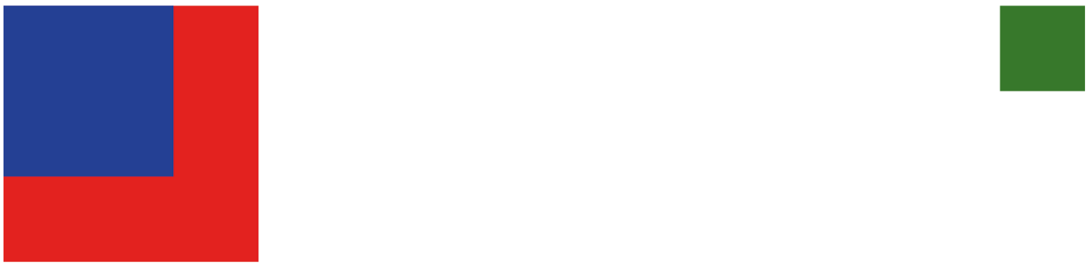


Figure 7-16. *The rendered result, showing the green box at the extreme right of the document*

The green box is now all the way to the right side of the document. This is because none of the green box's ancestor elements are positioned. When this happens, the element is positioned relative to the initial containing block. This is the document's root element.

To fix this, we can add `position: relative` to the outer red box. Once we do this, the green box now remains positioned inside the outer red box, as shown in Figure 7-17.

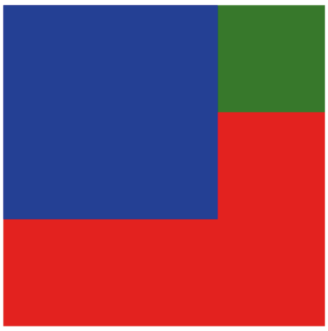


Figure 7-17. *The green box has changed position*

Now the green box is absolutely positioned, relative to the outer red box. This is because the red box is the closest positioned ancestor. If we were to add `position: relative` to the blue box, the green box would be positioned relative to that box instead because it becomes the closest positioned ancestor, as shown in Figure 7-18.

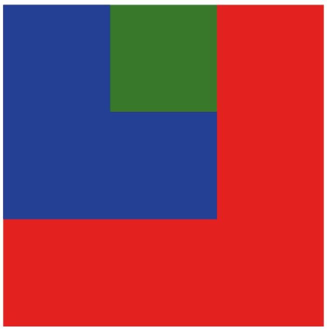


Figure 7-18. *The green box has a new closest positioned ancestor – the blue box*

Fixed Positioning

Like absolute positioning, fixed positioning removes the element from the document's flow. It is applied by setting the `position` property to `fixed`. Its position is determined by setting the `top`, `right`, `bottom`, and `left` properties.

The difference is that for a fixed positioned element, these offsets are always relative to the viewport instead of a positioned ancestor element. This means that even if the page is scrolled, a fixed element will remain in the same position. This is useful, for example, for a fixed header or navigation bar.

A block element with a `position` of `static` or `relative` will, by default, take up the full width of its container. However, if an element is given a position of `absolute` or `fixed`, this will no longer be the case. It will only be as wide as it needs to be to fit its content. This can usually be solved by adding `width: 100%` to the element if the full-width behavior is still desired.

Sticky Positioning

Setting an element's `position` property to `sticky` acts as a combination of relative and fixed positioning. The element acts as a relatively positioned element, scrolling with the document. When the element reaches a specified point, it turns into a fixed element. This point is specified via a `top`, `right`, `bottom`, or `left` value.

Z-index and Stacking Contexts

When elements have a `position` of `relative`, `fixed`, `absolute`, or `sticky`, they can partially cover other elements. This may not always behave the way you want. For example, suppose you have a page with a fixed header. You want to show a semi-transparent overlay to cover the whole page. The code for this is in Listing 7-16.

Listing 7-16. An example of a z-index issue

```
<style>
  .header {
    background-color: red;
    color: white;
    height: 1rem;
    left: 0;
    padding: 1rem;
    position: fixed;
    top: 0;
    width: 100%;
  }

  .body {
    margin-top: 3.5rem;
  }

  .overlay {
    background-color: rgba(0, 0, 0, 0.5);
    height: 100%;
    left: 0;
    position: absolute;
    top: 0;
    width: 100%;
  }
</style>

<div class="overlay"></div>
<div class="container">
  <header class="header">Header</header>
  <div class="body">Some other page content</div>
</div>
```

The result is shown in Figure 7-19. The overlay covers the body of the page, but not the header.

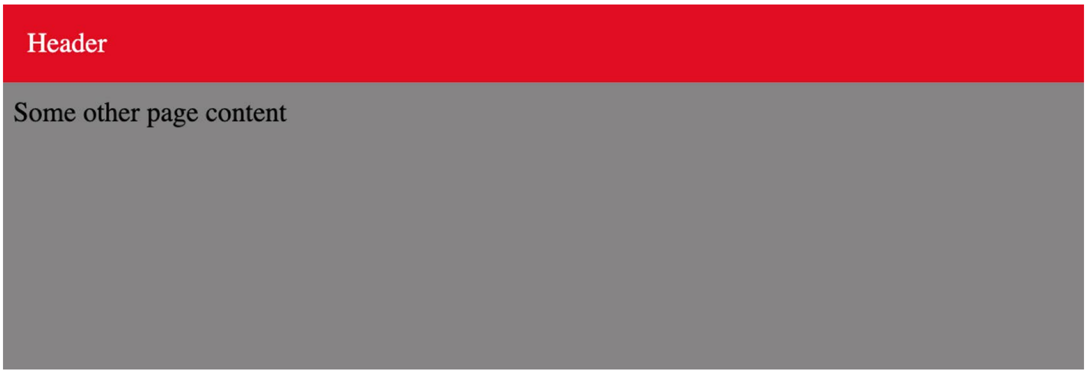


Figure 7-19. *The header is above the overlay*

This can be solved by giving the overlay a `z-index` value. The `z-index` property determines the stacking order of elements along the z-axis; it determines which element is on top of which. The `z-index` property is a relative measure that can take any numeric value. Items with a higher `z-index` will appear above those with a lower one.

In Listing 7-17, we'll give the overlay a `z-index` of `1`.

Listing 7-17. Adding a z-index

```
<style>
  .header {
    background-color: red;
    color: white;
    height: 1rem;
    left: 0;
    padding: 1rem;
    position: fixed;
    top: 0;
    width: 100%;
  }

  .body {
    margin-top: 3.5rem;
  }

  .overlay {
    background-color: rgba(0, 0, 0, 0.5);
    height: 100%;
    left: 0;
    position: absolute;
    top: 0;
    width: 100%;
    z-index: 1;
  }
</style>
```

```
<div class="overlay"></div>
<div class="container">
  <header class="header">Header</header>
  <div class="body">Some other page content</div>
</div>
```

As Figure 7-20 shows, now the overlay appears above the header.

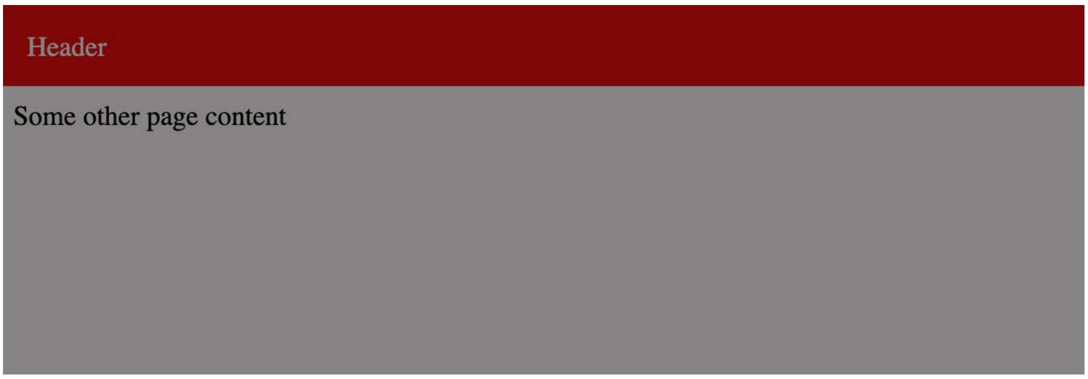


Figure 7-20. *The overlay now covers all of the content*

Stacking Contexts

As it turns out, z-index doesn't control an element's z-axis ordering globally within the entire document. It only controls the z-index relative to other elements within a group called a stacking context.

By default, there is one stacking context, formed by the root of the document (the `html` element). Within the document, there are rules for certain other elements that will create a new stacking context:

- Elements that have a `position` other than `static` and a `z-index` other than `auto`
- Elements with an `opacity` less than 1
- Elements that are children of a flex or grid layout and a `z-index` other than `auto`

There are other elements that create a new stacking context, but these are the most common ones. If an element doesn't have a `z-index` value, there are certain stacking rules that are applied within a given stacking context. The `z-index` ordering, from bottom to top, is

- The background and borders of the element that creates the stacking context
- Descendant elements of the element that creates the stacking context that have a `position` of `static`
- Descendant elements of the element that creates the stacking context that have a `position` value other than `static`

These rules, combined with explicitly set `z-index` values, determine the final stacking order of elements within a stacking context. Let's walk through a series of examples that illustrate `z-index` and stacking contexts. Consider the layout implemented in Listing 7-18.

Listing 7-18. A stacking context example

```
<style>
  .container-1 {
    background-color: red;
    width: 10rem;
    height: 10rem;
    position: relative;
    z-index: 100;
  }

  .container-2 {
    background-color: blue;
    width: 10rem;
    height: 10rem;
    position: relative;
    z-index: 100;
  }
  .inner-1 {
    background-color: green;
    width: 5rem;
    height: 5rem;
    position: relative;
    top: 7.5rem;
  }

  .inner-2 {
    background-color: orange;
    width: 5rem;
    height: 5rem;
    position: relative;
    top: -2.5rem;
  }
</style>

<div class="container-1">
  <div class="inner-1"></div>
</div>
<div class="container-2">
  <div class="inner-2"></div>
</div>
```

The resulting layout is shown in Figure 7-21.

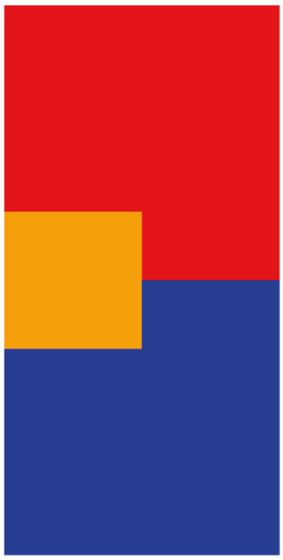


Figure 7-21. *The rendered layout*

Note that the two container elements both have their `position` properties set to `relative` and their `z-index` properties set to `100`. This means that each of these elements creates a new stacking context.

The two inner boxes – the green one and the orange one – have been positioned so that they are on top of each other. The orange box is on top. The green one isn't visible because it's underneath the orange one. Figure 7-22 shows a conceptual side cross-section view of the elements (dotted lines indicate stacking contexts).

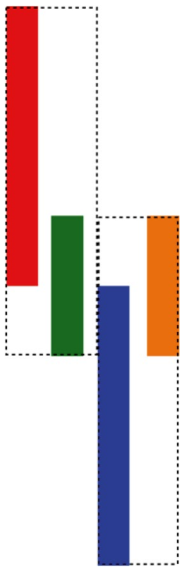


Figure 7-22. *A cross-section view*

Suppose we want the green box to be on top of the orange one. You might try adjusting the green box's `z-index` to be `200`, which is higher than all the other elements. However, the result is unchanged, and we'll see the same result as shown in Figure 7-21.

There is one difference, though. The cross-section has changed slightly, as shown in Figure 7-23.

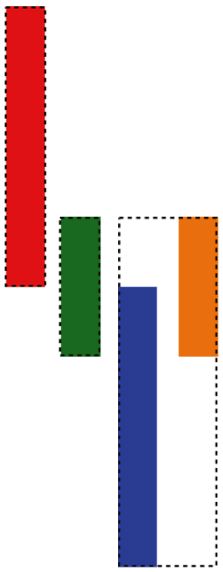


Figure 7-23. *The changed cross-section*

By setting the `z-index` property on the relatively positioned green box, we've created a new stacking context rooted at the green box. Even if you changed the `z-index` of the green box to something lower than the other elements, like `50`, it would still appear above the red box because it's in a higher stacking context.

Stacking context and `z-index` issues can be difficult to debug. Understanding how stacking contexts work, and how new ones are formed, is critical to solving these bugs when they come up.

Floats

You can use the `float` property to move an element to the left or right side, with text and other inline content flowing around it. Listing 7-19 shows a simple example of floating an element to the right.

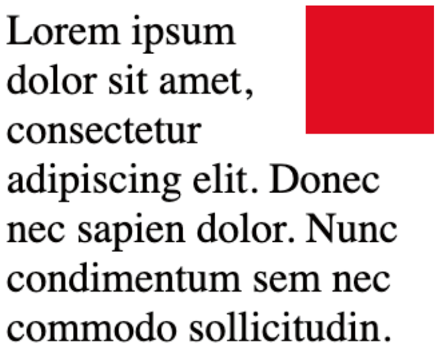
Listing 7-19. Floating an element to the right

```
<style>
  .container {
    width: 10rem;
  }

  .floating {
    background-color: red;
    float: right;
    height: 3rem;
    width: 3rem;
  }
</style>

<div class="container">
  <div class="floating"></div>
  Lorem ipsum dolor sit amet, consectetur
  adipiscing elit. Donec nec sapien
  dolor. Nunc condimentum sem nec
  commodo sollicitudin.
</div>
```

The result is shown in Figure 7-24. The red box floats to the right, and the text flows around it.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec nec sapien dolor. Nunc condimentum sem nec commodo sollicitudin.

Figure 7-24. *The floated red box and the text content that flows around it*

You can set the `float` property to `left` or `right`, or if you need to take text direction into account (left-to-right vs. right-to-left languages), you can use the logical `inline-start` or `inline-end` values.

When the `float` property is applied to an element, it is removed from the flow of the document. It then "floats" to the left or right, stopping when it reaches the edge of the containing element or another floated element. In the previous example, the red box moved to the right edge of the container. Listing 7-20 has another float example, this time using two floated elements.

Listing 7-20. Two floating elements

```
<style>
  .container {
    width: 10rem;
  }

  .floating,
  .floating-2 {
    float: right;
    height: 3rem;
    width: 3rem;
  }
  .floating {
    background-color: red;
  }

  .floating-2 {
    background-color: blue;
  }
</style>

<div class="container">
  <div class="floating"></div>
  <div class="floating-2"></div>
  Lorem ipsum dolor sit amet, consectetur
  adipiscing elit. Donec nec sapien
  dolor. Nunc condimentum sem nec
  commodo sollicitudin.
</div>
```

The result, two floating boxes, is shown in Figure 7-25.

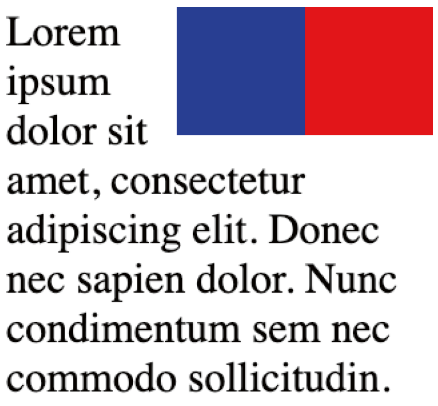


Figure 7-25. *The two floating boxes*

First, the red box is floated right, to the edge of the container. Next, the blue box is also floated right, stopping when it reaches the edge of the floated red box.

Clearing Floats

The `clear` property can be used on an element to indicate that it can't be alongside a floated element in a given direction. The `clear` property can be `none` (the default), `left`, `right`, `both`, `inline-start`, or `inline-end`.

If an element is cleared in a given direction, and there is a floated element there, the element will be moved so that it is below the floated element. Consider Listing 7-21, where there is a floated element on each side. We then use `clear: right` on the content.

Listing 7-21. Clearing floats

```
<style>
  .container {
    width: 10rem;
  }

  .floating {
    background-color: red;
    float: right;
    height: 3rem;
    width: 3rem;
  }

  .floating-2 {
    background-color: blue;
    float: left;
    height: 5rem;
    width: 3rem;
  }

  .content {
    clear: right;
  }
</style>

<div class="container">
  <div class="floating"></div>
  <div class="floating-2"></div>
  <div class="content">
    Lorem ipsum dolor sit amet, consectetur
    adipiscing elit. Donec nec sapien
    dolor. Nunc condimentum sem nec
    commodo sollicitudin.
  </div>
</div>
```

The result is shown in Figure 7-26.

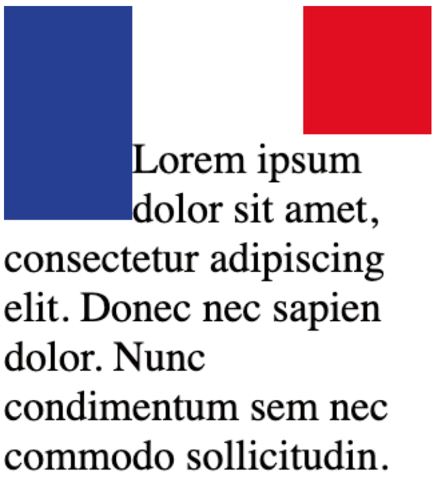


Figure 7-26. *The floating elements and cleared content*

The content has been moved to below the right-floated red box because we specified `clear: right`. The content is not cleared to the left, so the blue box is still allowed to float there, and the text wraps around it.

Width and Height

Most of the time, when specifying an element's `width` and `height`, you'll probably use a unit like `px`, `em`, or `rem`. However, there are some additional special values that can be applied to width and height.

Intrinsic vs. Extrinsic Size

When you specify a specific width or height like `250px` or `5rem`, you're setting an *extrinsic size*. The extrinsic size does not take the element's content into account. Rather, it uses the explicitly specified values.

An element's *intrinsic size* is the size an element would normally take up, given its content. While extrinsic sizes can be specified for block elements, inline elements only use their intrinsic size.

The `min-content` and `max-content` Keywords

Instead of a specific value, you can set an element's `height` or `width` using the `min-content` or `max-content` keywords.

When you use `min-content`, the element is set to the smallest possible size it can be without the content overflowing. For an element's width, this generally means the element's width will be limited to the width of the longest word (assuming normal word wrapping behavior is used). Listing 7-22 shows an example of using `min-content`.

Listing 7-22. Setting the width to `min-content`

```
<style>
  .text {
    width: min-content;
    background: skyblue;
  }
</style>

<div class="text">
    Hello world, how are
    you doing today?
</div>
```

The result is shown in Figure 7-27.

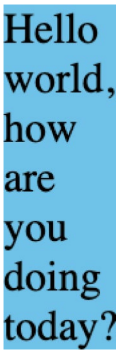


Figure 7-27. *The element sized with a `width` of `min-content`*

The element is only as wide as is necessary to render the text inside it without the text overflowing the container.

When you use `max-content`, the element is set to the largest possible size it can be so that the text doesn't wrap. This is shown in Listing 7-23.

Listing 7-23. Setting the width to `max-content`

```
<style>
    .text {
        width: max-content;
        background: skyblue;
    }
</style>

<div class="text">
    Hello world, how are
    you doing today?
</div>
```

The result is shown in Figure 7-28.

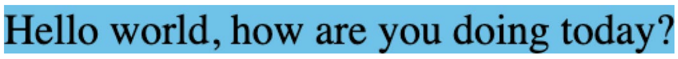


Figure 7-28. *The element sized with a `width` of `max-content`*

Now the element is wide enough to fit the full sentence, so that the text content doesn't wrap.

Using Multiple Columns

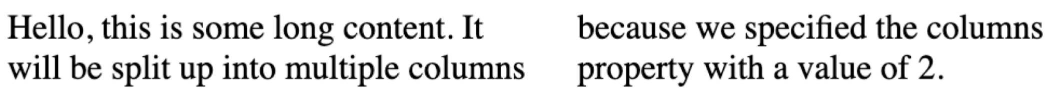
You can split an element's content to flow into multiple columns with the `columns` property. Listing 7-24 has an example of this.

Listing 7-24. Using the `columns` property

```
<style>
    .content {
        columns: 2;
        width: 32rem;
    }
</style>

<div class="content">
    Hello, this is some long content.
    It will be split up into multiple columns
    because we specified the columns property
    with a value of 2.
</div>
```

The resulting column layout is shown in Figure 7-29.



Hello, this is some long content. It will be split up into multiple columns because we specified the columns property with a value of 2.

Figure 7-29. *The text split up over two columns*

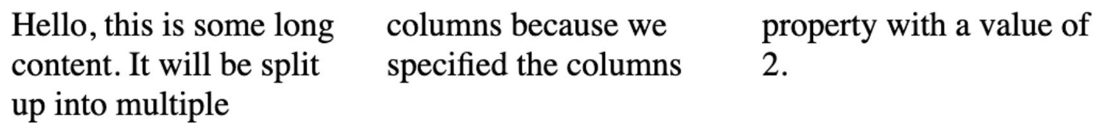
Instead of a number of columns, you can also specify the desired column width, and the number of columns will be calculated automatically, as shown in Listing 7-25.

Listing 7-25. Specifying a column width

```
<style>
    .content {
        columns: 10rem;
        width: 32rem;
    }
</style>
```

```
<div class="content">
    Hello, this is some long content.
    It will be split up into multiple columns
    because we specified the columns property
    with a value of 2.
</div>
```

The text will be split into three 10rem columns, as shown in Figure 7-30.



Hello, this is some long content. It will be split up into multiple columns because we specified the columns property with a value of 2.

Figure 7-30. *The three-column layout*

Summary

- Padding is the spacing between an element's content and its border.
- Margin is the spacing between an element's border and other elements.
- An element is said to be positioned if it has a `position` property set to something other than `static`.
- Elements can have a position of `static`, `relative`, `absolute`, `fixed`, or `sticky`.
    - Statically positioned elements flow normally.
    - Relatively positioned elements are positioned relative to their normal position in the document.
    - Absolutely positioned elements are positioned relative to their nearest positioned ancestor and are removed from the normal document flow.
    - Fixed positioned elements remain fixed with respect to the viewport.
    - Sticky positioned elements are a hybrid between relative and fixed positioning.

- The `z-index` property controls the vertical stacking order of an element within a given stacking context.
- The `float` property allows an element to be floated to the left or right sides of its container, and other inline content flows around it.
- The `min-content` keyword sizes an element to its smallest possible size without overflowing.
- The `max-content` keyword sizes an element to its largest possible size without wrapping.
- The `columns` property lets you break up the flow of a container into multiple columns. A column count or column width can be specified.

CHAPTER 8

Transforms

CSS provides a set of transformations that can be applied to an element's appearance. For example, an element can be rotated in 2D or 3D space, scaled, skewed, or translated (moved from its original position). Transforms can be used to create all kinds of interesting effects on their own and become more powerful when combined with transitions and animations, which we will cover in Chapter 9.

Transforms can be applied in two ways. You can use the `transform` property, specifying one or more transform functions, separated by a space. Each transform also has its own CSS property that you can use.

For example, if you want to rotate an element by 45 degrees, you can use the `transform` property (`transform: rotate(45deg)`) or the `rotate` property (`rotate: 45deg`).

The following sections will cover the most common types of transforms you can apply to an element.

The X-, Y-, and Z-axes

Many transforms can be performed along one or more axes. The coordinate system is as follows:

- *X-axis*: Goes from left to right across the screen
- *Y-axis*: Goes from top to bottom vertically along the screen
- *Z-axis*: 3D axis, goes from the "surface" of the page out toward you.

These three axes are visualized in Figure 8-1.

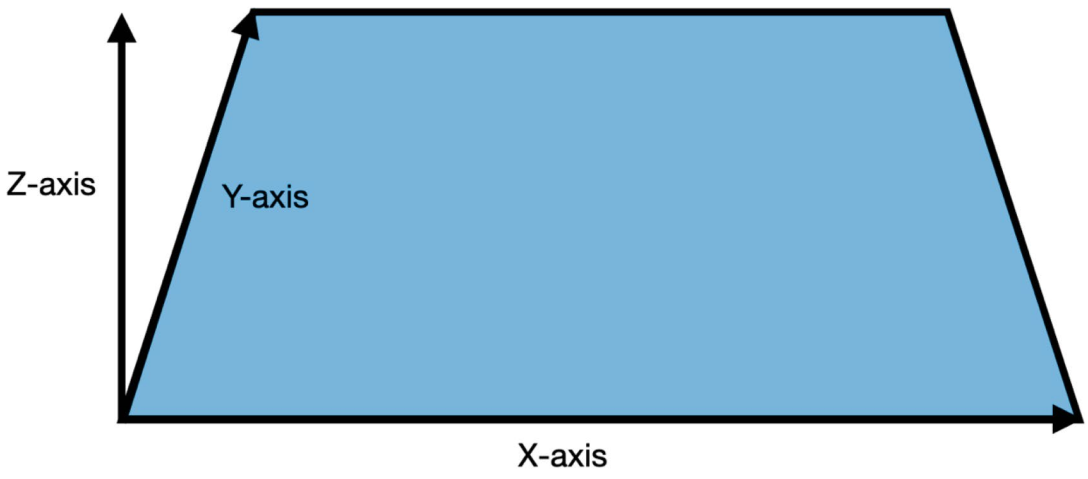


Figure 8-1. *The three axes of the coordinate system*

Perspective

Let's start by discussing the `perspective` transform. This is necessary when using certain types of 3D transforms. It defines how "far away" the object will appear from the user, as if the screen has depth. The lower the number, the closer the object will appear. Used by itself, `perspective` has no visible effect. But when combined with certain transforms, it can greatly affect the final result.

Rotate

The `rotate` transform rotates an element around a given axis. It takes the angle to rotate by as an argument. The angle can be whole or fractional and is given in one of several units:

- `deg`: Degrees. A full circle is `360deg`.
- `grad`: Gradians. A full circle is `400grad`.
- `rad`: Radians. A full circle is approximately 2π radians, or approximately `6.28rad`.
- `turn`: Number of turns. A full circle is `1turn`.

A positive angle rotates the element clockwise, and a negative angle rotates it counterclockwise. Note that an element can be rotated by more than one full circle.

Rotation Axis

The rotation axis defines which direction the element is rotated in. The element is rotated *around* the specified axis.

Rotation Origin

A rotation also has an origin. This is the point around which the element is rotated. By default, this is the center of the element, as shown in Figure 8-2.

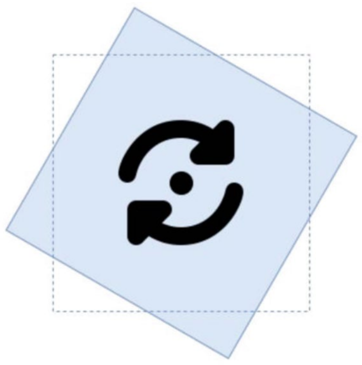


Figure 8-2. *The default rotation origin, the center of the element*

However, you can specify a different origin with the `transform-origin` property. This will change the rotation point, as shown in Figure 8-3. Here, the element will be rotated around its top-left corner.

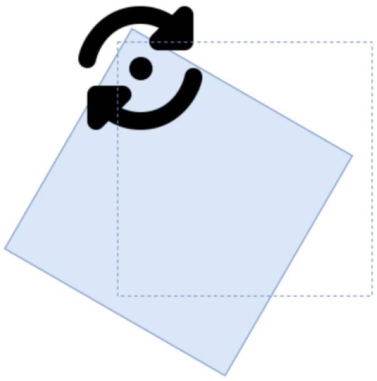


***Figure 8-3.** A different rotation origin*

The `transform-origin` property is specified as one, two, or three values. These values correspond to the X, Y, and Z offsets to use for the origin point. These can be size values such as `10px` or `25%` or one of the keywords `left`, `center`, `right`, `top`, or `bottom`.

Rotating Around the Z-axis

By default, when using the `rotate` transform, the element is rotated around the Z-axis. If you want to be more explicit, you can also use the `rotateZ` transform. Listing 8-1 shows an example of a Z-axis rotation.

Listing 8-1. Rotating around the Z-axis

```
<style>
    .rotate {
        width: 100px;
        height: 100px;
        background: skyblue;
        transform: rotate(45deg);
        margin: 50px;
    }
</style>

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

The element is rotated around the Z-axis by 45 degrees, as shown in Figure 8-4.

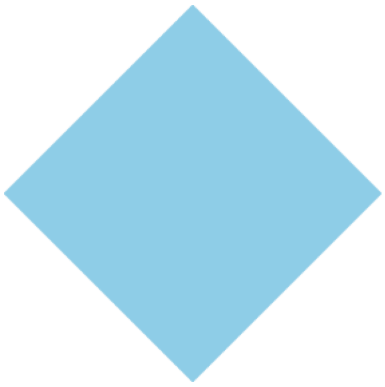


Figure 8-4. *The rotated element*

You can also rotate an element by using the `rotate` CSS property directly, instead of the transform function. This is shown in Listing 8-2.

Listing 8-2. Using the `rotate` property

```
<style>
    .rotate {
        width: 100px;
        height: 100px;
        background: skyblue;
        rotate: 45deg;
        margin: 50px;
    }
</style>

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

Rotating Around the X-axis

The `rotateX` transform rotates an element around the horizontal X-axis. Used on its own, the 3D effect isn't noticeable, as demonstrated in Listing 8-3.

Listing 8-3. Rotating around the X-axis

```
<style>
    .rotate {
        width: 100px;
        height: 100px;
        background: skyblue;
        transform: rotateX(45deg);
        margin: 50px;
    }
</style>

<div class="rotate">Rotate me!</div>
```

The element is rotated around the X-axis, but it still appears as a flat rectangle, as shown in Figure 8-5. The text appears distorted.

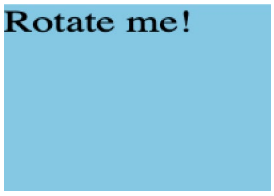


Figure 8-5. *No rotation effect is observed*

To get the 3D rotation effect we want, we need to use the `perspective` transform to activate the 3D space. Here, we'll use a `perspective` of `200px`, but you can experiment with the `perspective` value to fine-tune the appearance of the rotation transform.

Listing 8-4 adds the necessary perspective.

Listing 8-4. Adding perspective to the `rotateX` transform

```
<style>
    .rotate {
        width: 100px;
        height: 100px;
        background: skyblue;
        transform: perspective(200px) rotateX(45deg);
        margin: 50px;
    }
</style>

<div class="rotate">Rotate me!</div>
```

Now you can see the 3D transform effect, as shown in Figure 8-6.

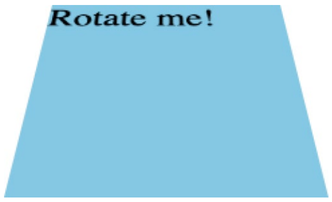


Figure 8-6. *The rotation with perspective*

Rotating Around the Y-axis

You can rotate an element around the Y-axis with the `rotateY` transform. An example of this is given in Listing 8-5. The example also adds `200px` of perspective as we did in Listing 8-4.

Listing 8-5. Rotating around the Y-axis

```
<style>
    .rotate {
        width: 100px;
        height: 100px;
        background: skyblue;
        transform: perspective(200px) rotateY(45deg);
        margin: 50px;
    }
</style>

<div class="rotate">Rotate me!</div>
```

The element is rotated around the Y-axis, as shown in Figure 8-7.

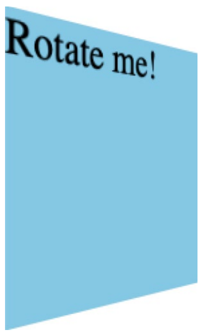


Figure 8-7. *The rotated element*

Rotating in Three Dimensions

The `rotate3d` transform rotates an element around an arbitrary axis in 3D space. This is done by defining a direction vector in the 3D coordinate system. The vector is defined by specifying the component of the vector in each direction (X, Y, and Z) as the coordinate value. The element is then rotated around that vector by the given angle.

Listing 8-6 uses the `rotate3d` transform to rotate the element in the X and Y directions.

Listing 8-6. Using the `rotate3d` transform

```
<style>
    .rotate {
        width: 100px;
        height: 100px;
        background: skyblue;
        transform: perspective(200px) rotate3d(1, 1, 0, 45deg);
        margin: 50px;
    }
</style>

<div class="rotate">Rotate me!</div>
```

The resulting transformed element is shown in Figure 8-8.

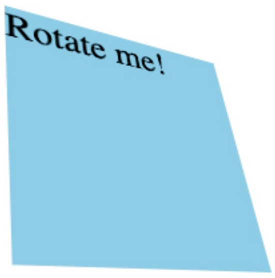


Figure 8-8. *The element rotated in the X and Y directions*

Translate

The next type of transform we'll look at is translation. Translating an element means moving it from its original position.

The `translate` transform moves an element in 2D space. It takes one or two arguments, corresponding to the distance along the X-axis and Y-axis, respectively. The flow of the document is not affected by translating an element; a blank space is left where the element's original position was.

Listing 8-7 shows an example of moving an element with the `translate` transform.

Listing 8-7. Translating an element

```
<style>
  div {
    width: 5rem;
    height: 5rem;
    display: inline-block;
  }

  .one {
    background: orangered;
  }

  .two {
    background: rebeccapurple;
    transform: translate(2rem, 2rem);
  }
  .three {
    background: skyblue;
  }
</style>

<div class="one"></div>
<div class="two"></div>
<div class="three"></div>
```

This example renders three boxes, and the purple box in the middle is moved by `2rem` in the X and Y directions, as shown in Figure 8-9.

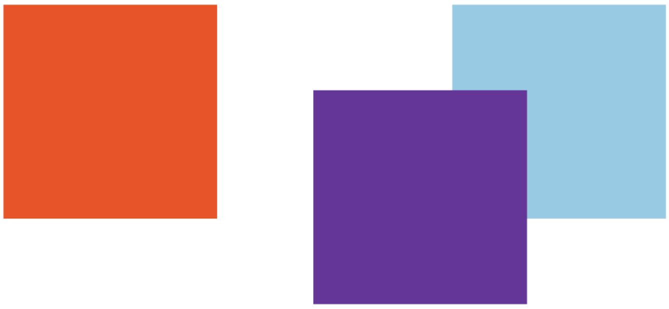


***Figure 8-9.** The translated element*

The purple box in Figure 8-9 was translated `2rem` to the left and `2rem` down, but its original position leaves a “hole” in the layout. The red and blue boxes do not change their position to compensate for the translated element.

If you only want to translate in one direction, you can use the `translateX` or `translateY` transforms. The effect is the same as when using the `translate` function. That is:

- `translateX(1rem)` is equivalent to `translate(1rem, 0)`.
- `translateY(1rem)` is equivalent to `translate(0, 1rem)`.

Translating Along the Z-axis

The `translateZ` transform moves an element along the Z-axis. It has the effect of moving the element closer or farther away from the user’s perspective. It only has a visible effect when used with the `perspective` transform.

Listing 8-8 shows an example of translating along the Z-axis.

Listing 8-8. Translating along the Z-axis

```
<style>
  div {
    width: 5rem;
    height: 5rem;
    display: inline-block;
  }

  .one {
    background: orangered;
  }

  .two {
    background: rebeccapurple;
    transform: perspective(200px) translateZ(2rem);
  }

  .three {
    background: skyblue;
  }
</style>

<div class="one"></div>
<div class="two"></div>
<div class="three"></div>
```

The result is shown in Figure 8-10.

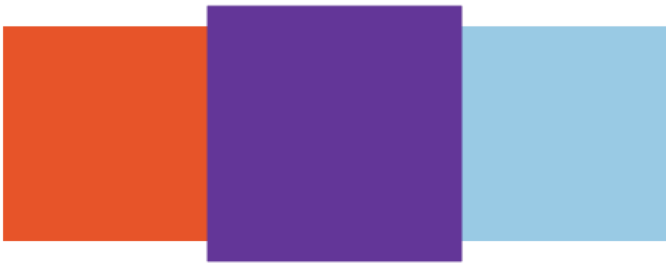


Figure 8-10. *The element translated along the Z-axis*

As Figure 8-10 shows, the purple box has the appearance of being moved closer to the user.

Translating in an Arbitrary Direction with `translate3d`

Like `rotate3d`, `translate3d` allows you to specify a vector in 3D space. The element is then translated along that vector. The three arguments define the X, Y, and Z components of the vector. An example of this is shown in Listing 8-9.

Listing 8-9. Translating with the `translate3d` transform

```
<style>
  div {
    width: 5rem;
    height: 5rem;
    display: inline-block;
  }

  .one {
    background: orangered;
  }

  .two {
    background: rebeccapurple;
    transform: perspective(200px) translate3d(1rem, 2rem, 3rem);
  }

  .three {
    background: skyblue;
  }
</style>

<div class="one"></div>
<div class="two"></div>
<div class="three"></div>
```

The resulting transform is shown in Figure 8-11.

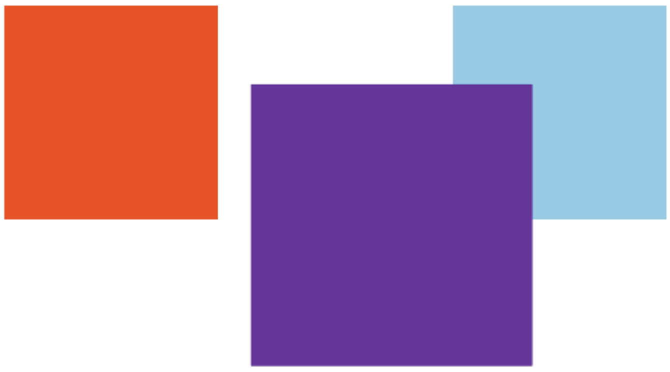


Figure 8-11. *The translated element*

Translating with the `translate` CSS Property

Instead of using the `transform` property with the `translate` function, you can also translate an element with the top-level `translate` CSS property. Listing 8-10 shows an example of this.

Listing 8-10. Translating with the `translate` property

```
<style>
  div {
    width: 5rem;
    height: 5rem;
    display: inline-block;
  }

  .one {
    background: orangered;
  }

  .two {
    background: rebeccapurple;
    translate: 2rem 2rem;
  }
  .three {
    background: skyblue;
  }
</style>

<div class="one"></div>
<div class="two"></div>
<div class="three"></div>
```

The result is the same as when using the transform property, as Figure 8-12 shows.

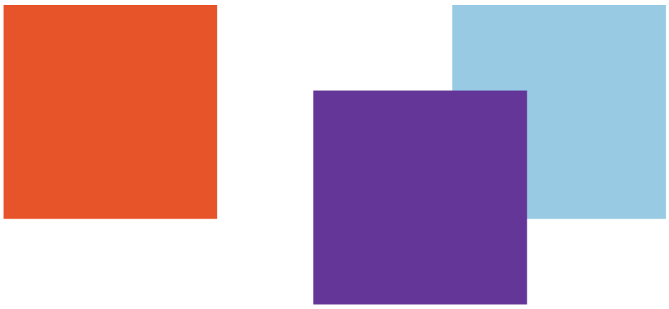


Figure 8-12. *The translated element*

Scale

The `scale` transform alters the size of an element, scaling its contents as it grows or shrinks. The `scale` transform function takes a scaling factor, which is a multiple of the element's original size. A scaling factor of 1 is the original size, 2 is twice the original size, and 0.5 is half the original size.

The `scale` transform can take one or two arguments. If one argument is given, it applies the given scaling factor in both the X and Y directions. If two arguments are given, the first argument is the X scaling factor, and the second argument is the Y scaling factor. Listing 8-11 shows two example usages of the `scale` transform.

Listing 8-11. Scaling elements

```
<style>
    .scale-1 {
        transform: scale(2);
        background: skyblue;
        width: 150px;
        margin: 100px;
    }

    .scale-2 {
        transform: scale(0.5, 3);
        background: orangered;
        width: 150px;
        margin: 50px;
    }
</style>

<div class="scale-1">scale(2)</div>
<div class="scale-2">scale(0.5, 3)</div>
```

The extra margin is needed to prevent the scaled element from extending past the edge of the screen. The resulting scaled elements are shown in Figure 8-13.

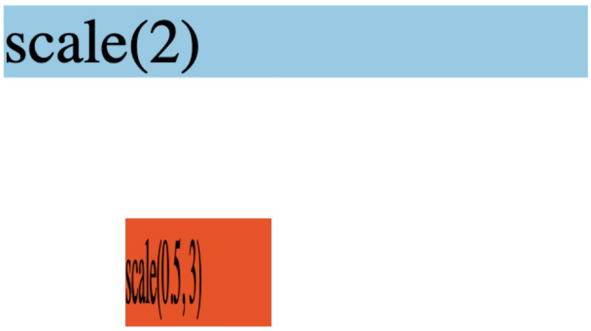


Figure 8-13. *The scaled elements*

Notice that since the orange box is scaled differently in the X and Y directions, the text appears distorted. Since the blue box is scaled uniformly, there is no text distortion.

By default, the scaling functions perform the transform starting at the center of the element. This can be changed by giving a value for the `transform-origin` property. Also, note that scaling an element will not cause its container to grow to fit the new size. The flow of the document is not affected. This behavior is shown in Figure 8-14.

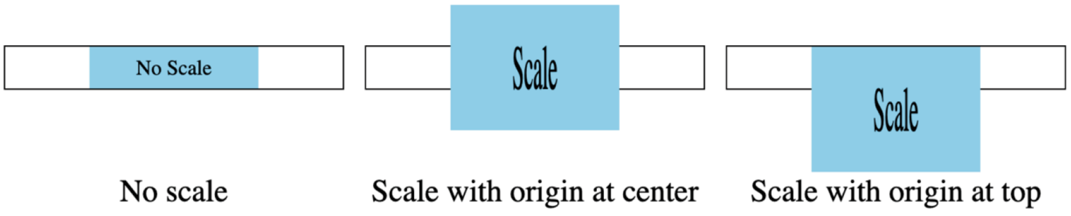


Figure 8-14. *The different scale behavior depending on the transform-origin*

Scaling with the `scale` CSS Property

Like with other transforms, there is also a top-level `scale` CSS property that you can use. The value can be one or two scaling factors. If only one value is given, it is used in both the X and Y direction. If two values are given, the first value is the X scale factor, and the second is the Y scale factor. Listing 8-12 shows an example of scaling an element using this property.

Listing 8-12. Scaling with the `scale` property

```
<style>
    .scale {
        width: 200px;
        background: skyblue;
        scale: 2 4;
        margin: 100px;
    }
</style>

<div class="scale">Scale</div>
```

The scaled element is shown in Figure 8-15.

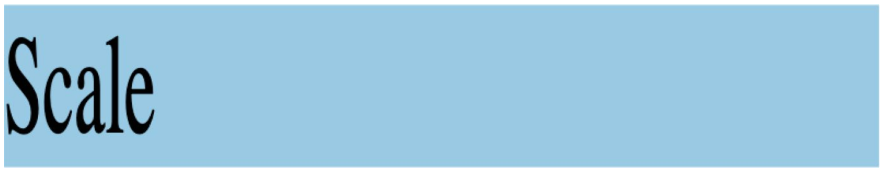


Figure 8-15. *The scaled element*

Skew

The skew transform distorts an element by a given angle in the X and Y directions. Like the rotate transform, the angle can be given in one of several different units such as deg or rad. Listing 8-13 shows an example of skewing an element.

Listing 8-13. An example of the skew transform

```
<style>
  .skew {
    background: skyblue;
    transform: skew(45deg, 20deg);
    width: 10rem;
    font-size: 2rem;
    text-align: center;
    margin: 5rem;
  }
</style>

<div class="skew">Hello world!</div>
```

The skewed element is shown in Figure 8-16.

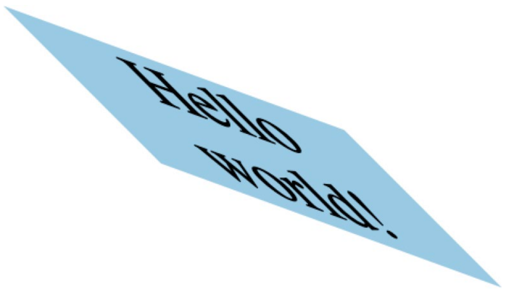


Figure 8-16. *The skewed element*

If you want to skew an element in one direction only, you can use the skewX and skewY transform functions.

Applying Multiple Transforms

The `transform` property can have more than one transform function, separated by spaces. Listing 8-14 shows an example of applying both a translate and scale transform.

Listing 8-14. Applying multiple transforms

```
<style>
  div {
    width: 5rem;
    height: 5rem;
    display: inline-block;
  }

  .one {
    background: orangered;
  }

  .two {
    background: rebeccapurple;
    transform: translate(2rem, 2rem) scale(1.5);
  }

  .three {
    background: skyblue;
  }
</style>

<div class="one"></div>
<div class="two"></div>
<div class="three"></div>
```

Figure 8-17 shows the element with both transforms applied. The purple box is translated down and to the right and is also scaled by a factor of 1.5.

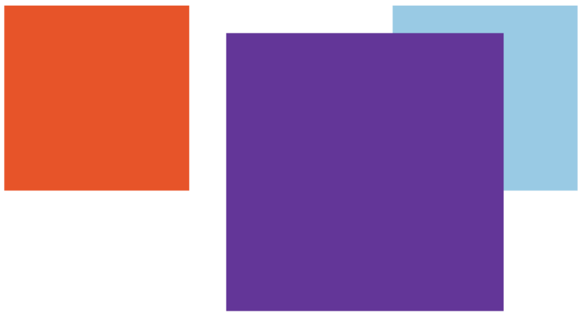


Figure 8-17. *The translated and scaled element*

Things get a little tricky when rotation and translation are involved, though. This is when the order of the transforms really matters. When you rotate an element, its X/Y coordinate system rotates with it, as Figure 8-18 shows.

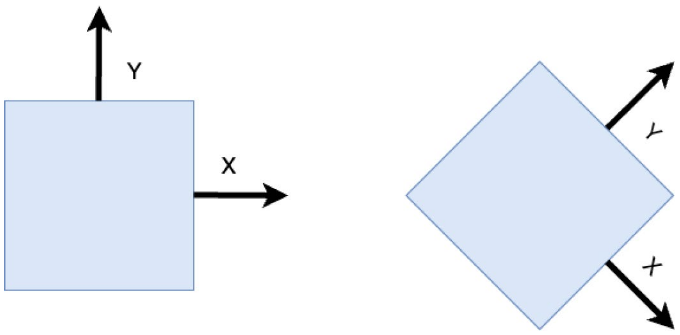


Figure 8-18. *The coordinate system rotates with the element*

This means that the final position of the element is different depending on whether you rotate before or after you translate. Listing 8-15 shows the difference.

Listing 8-15. Rotating and translating

```
<style>
    .container {
        border: 1px solid black;
        width: 100px;
        height: 100px;
        margin: 100px;
    }
    .transform-1 {
        background: skyblue;
        transform: translateX(50px) rotate(45deg);
        width: 100px;
        height: 100px;
    }

    .transform-2 {
        background: skyblue;
        transform: rotate(45deg) translateX(50px);
        width: 100px;
        height: 100px;
    }
</style>

<div class="container">
    <div class="transform-1"></div>
</div>

<div class="container">
    <div class="transform-2"></div>
</div>
```

If you compare the elements' positions in Figure 8-19, you'll notice that they end up in slightly different places due to the different order of the transforms.

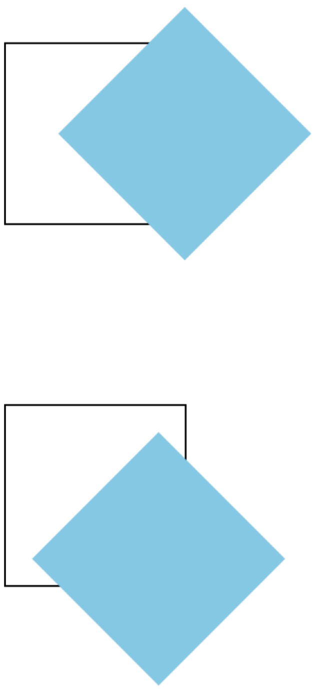


Figure 8-19. *The transformed elements*

You can even specify multiple transforms of the same type. Listing 8-16 translates, rotates, then translates again.

Listing 8-16. Using multiple transforms of the same type

```
<style>
  .container {
    border: 1px solid black;
    width: 100px;
    height: 100px;
    margin: 100px;
  }

  .transform {
    background: skyblue;
    transform: translateX(50px) rotate(45deg) translateX(100px);
    width: 100px;
    height: 100px;
  }
</style>

<div class="container">
  <div class="transform"></div>
</div>
```

Figure 8-20 shows the blue box's final position. We move it 100 pixels along the X-axis, then rotate it by 45 degrees. When we rotate the box, its coordinate system also rotates. When we apply the second `translateX` transform, the box moves along its rotated X-axis.

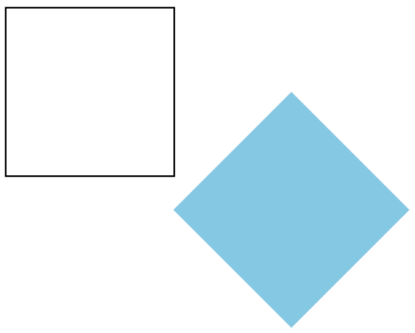


Figure 8-20. *Applying three transforms to the element*

Putting It All Together: Making a Cube

Let's apply what we learned about CSS transforms to make a cube shape. There is a container element for the cube, the cube itself, and one element for each of the six faces of the cube. Listing 8-17 has the initial markup and CSS.

Listing 8-17. The initial cube code

```
<style>
  .container {
    width: 10rem;
    height: 10rem;
    perspective: 500px;
    margin: 5rem;
  }
  .cube {
    position: relative;
    width: 10rem;
    height: 10rem;
    transform-style: preserve-3d;
    transform: rotate3d(1, 1, 0, 45deg);
  }

  .face {
    width: 10rem;
    height: 10rem;
    background: skyblue;
    border: 2px solid black;
    position: absolute;
    opacity: 0.5;
    text-align: center;
  }
</style>

<div class="container">
  <div class="cube">
    <div class="face top">Top</div>
    <div class="face bottom">Bottom</div>
    <div class="face left">Left</div>
    <div class="face right">Right</div>
    <div class="face front">Front</div>
    <div class="face back">Back</div>
  </div>
</div>
```

Listing 8-17 introduces a new property: `transform-style`. By default, an element's children are flattened to be on the same plane. This means they are "squashed" down to 2D space. We want to make a 3D cube, so that won't work here. Setting `transform-style` to `preserve-3d` will allow the cube's child elements to exist in 3D space.

We're also applying a 3D rotation to give us a better look at the cube's structure. The faces are made partially transparent, so we can see through to the other faces to better visualize the cube.

Figure 8-21 shows the result so far. All the cube's faces are lying flat, stacked on top of one another, since they are absolutely positioned. What we need to do is rotate each face, in 3D space, so that they are facing the correct way. Then we need to move them out from the center to form the cube.

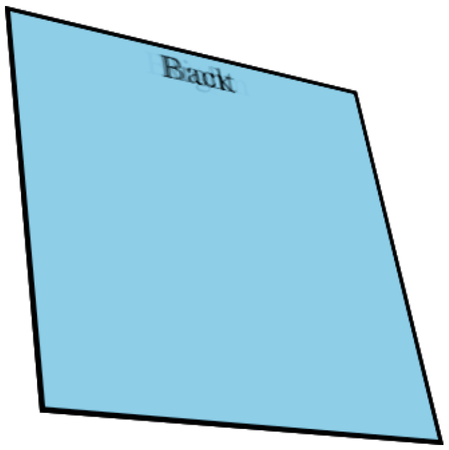


Figure 8-21. *The cube faces all stacked on top of each other*

Now we need to rotate the faces to their proper orientations. The front doesn't need to be rotated, since it's already facing forward. We need to make the following transforms:

- Rotate the back around the Y-axis by 180 degrees
- Rotate the left around the Y-axis by –90 degrees
- Rotate the right around the Y-axis by 90 degrees
- Rotate the top around the X-axis by 90 degrees
- Rotate the bottom around the X-axis by –90 degrees

Listing 8-18 applies these transforms to the cube code.

Listing 8-18. Rotating the cube faces

```
<style>
  .container {
    width: 10rem;
    height: 10rem;
  perspective: 500px;
  margin: 5rem;
}

.cube {
  position: relative;
  width: 10rem;
  height: 10rem;
  transform-style: preserve-3d;
  transform: rotate3d(1, 1, 0, 45deg);
}

.face {
  width: 10rem;
  height: 10rem;
  background: skyblue;
  border: 2px solid black;
  position: absolute;
  opacity: 0.5;
  text-align: center;
}

.back {
  transform: rotateY(180deg);
}

.left {
  transform: rotateY(-90deg);
}

.right {
  transform: rotateY(90deg);
}

.top {
  transform: rotateX(90deg);
}
```

```
  .bottom {
    transform: rotateX(-90deg);
  }
</style>

<div class="container">
  <div class="cube">
    <div class="face top">Top</div>
    <div class="face bottom">Bottom</div>
    <div class="face left">Left</div>
    <div class="face right">Right</div>
    <div class="face front">Front</div>
    <div class="face back">Back</div>
  </div>
</div>
```

Figure 8-22 shows the rotated cube faces.

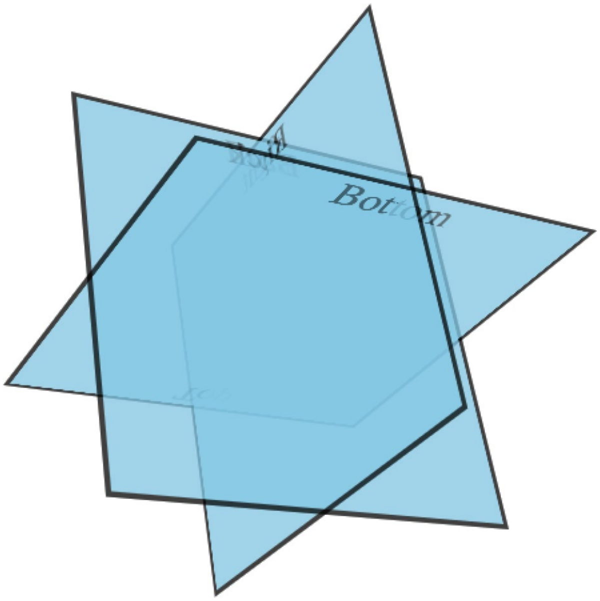


Figure 8-22. *The rotated cube faces*

Now all the faces are rotated properly, but they are still at the center of the cube. Since the cube's size is 10rem, and the faces are in the middle, each face must be moved out by 5rem in the proper direction:

- The front and back along the Z-axis
- The left and right along the X-axis
- The top and bottom along the Y-axis

Since the axes will change position after rotating, we need to do the translate first. Otherwise, the faces would end up the wrong position. Listing 8-19 has the final cube code.

Listing 8-19. Moving the faces out

```
<style>
  .container {
    width: 10rem;
    height: 10rem;
    perspective: 500px;
    margin: 5rem;
  }

  .cube {
    position: relative;
    width: 10rem;
    height: 10rem;
    transform-style: preserve-3d;
    transform: rotate3d(1, 1, 0, 45deg);
  }

  .face {
    width: 10rem;
    height: 10rem;
    background: skyblue;
    border: 2px solid black;
    position: absolute;
    opacity: 0.5;
    text-align: center;
  }
  .front {
    transform: translateZ(5rem);
  }

  .back {
    transform: translateZ(-5rem) rotateY(180deg);
  }

  .left {
    transform: translateX(-5rem) rotateY(-90deg);
  }

  .right {
    transform: translateX(5rem) rotateY(90deg);
  }

  .top {
    transform: translateY(-5rem) rotateX(90deg);
  }

  .bottom {
    transform: translateY(5rem) rotateX(-90deg);
  }
</style>

<div class="container">
  <div class="cube">
    <div class="face top">Top</div>
    <div class="face bottom">Bottom</div>
    <div class="face left">Left</div>
    <div class="face right">Right</div>
    <div class="face front">Front</div>
    <div class="face back">Back</div>
  </div>
</div>
```

This completes the cube, which is shown in Figure 8-23.

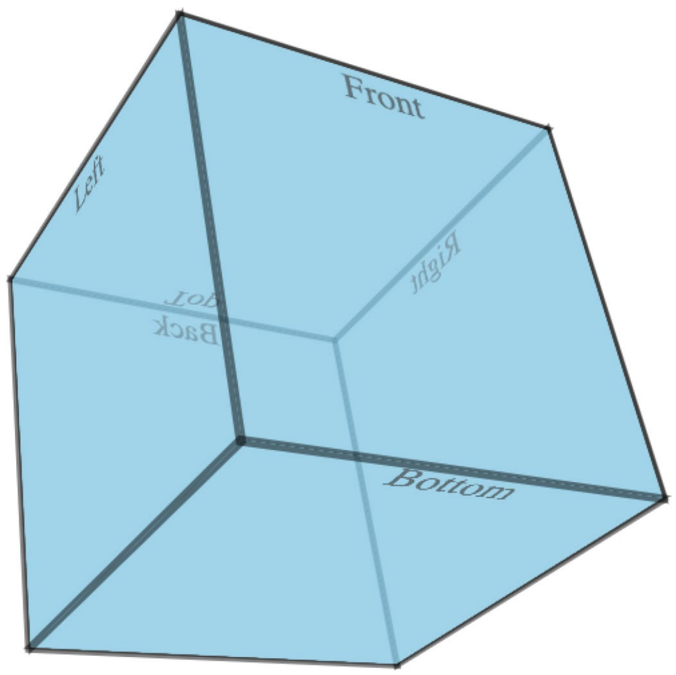


Figure 8-23. *The completed cube*

You can combine CSS transforms to create all kinds of 2D and 3D shapes.

Summary

- Elements can be transformed along the X-, Y-, and Z-axes.
- The `perspective` transform is necessary to see some 3D transforms.
- The `transform-origin` property specifies from what point within the element that a given transformation is applied.
- An element can be rotated with the `rotate` transform function or the `rotate` property.
- An element can be moved from its original position with the `translate` transform function or the `transform` property.

- An element can be resized with the `scale` transform function or the `scale` property.
- An element can be skewed with the `skew` transform function or the `skew` property.
- Multiple transforms can be applied to an element. When dealing with rotation, the order of the transform functions matters.

CHAPTER 9

Transitions and Animations

CSS transforms are useful on their own, but they are even more powerful when combined with transitions and animations.

Transitions

A CSS transition is a way of animating an element from one state to another. During the lifetime of a page, an element's style can change. For example, the user could hover over an element, triggering the `:hover` pseudo-class, which applies some different styling.

Or maybe a class is added to or removed from an element with JavaScript. In both cases, any style changes are applied immediately. Let's take the example of a hover state. Consider the styles for a button shown in Listing 9-1.

Listing 9-1. Some button styles

```
<style>
  button.fancy-button {
    background: blue;
  }

  button.fancy-button:hover {
    background: red;
    transform: scale(1.1);
  }
</style>

<button class="fancy-button">Fancy Button</button>
```

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_9

When the user hovers over this button with their mouse, two things will happen. The background color will immediately change from blue to red, and the button will immediately transform to a scale factor of 1.1. You can improve this experience by using a CSS transition to animate the style change. Listing 9-2 shows how to add a transition with the `transition` property.

Listing 9-2. Adding a transition

```
<style>
  button.fancy-button {
    background: blue;
    transition: 500ms;
  }

  button.fancy-button:hover {
    background: red;
    transform: scale(1.1);
  }
</style>

<button class="fancy-button">Fancy Button</button>
```

With the transition applied, the behavior will be different. Instead of immediately snapping to the new background color and size, the element will undergo an animated transition to the new color and scale over a period of 500 milliseconds. The styles are animated from the base style to the hover style.

The color will gradually change from blue to shades of purple and to the final color of red. At the same time, the size will animate from `scale(1)` to `scale(1.1)`, making the element grow in size over the course of the 500-millisecond transition.

You can apply more than one transition to an element. For example, suppose you want the background color to transition first, and then only after the color change is complete, then transition the `scale` transform.

The styles can be transitioned independently. Listing 9-2 only specified a transition duration, not a transition property, so all available properties were transitioned together – the color and scale.

To separate these transitions, each option given to the `transition` property can include the name of a property to transition, as well as its duration. Another number can be added to define a delay, so that the `scale` transform doesn't start transitioning until the background color transition is complete.

Listing 9-3 has an example of a multi-step transition. We want the following two transition stages:

- Transition the color over 500 milliseconds
- Transform the scale transform over 500 milliseconds, with a 500-millisecond delay

Listing 9-3. A multi-stage transition

```
<style>
  button.fancy-button {
    background: blue;
    transition: background-color 500ms,
                transform 500ms 500ms;
  }

  button.fancy-button:hover {
    background: red;
    transform: scale(1.1);
  }
</style>

<button class="fancy-button">Fancy Button</button>
```

When specifying a transition in this way, the first value is the name of the property to transition, the second property is the duration, and the third property is the delay. Now, when you hover over the fancy button, the color transitions from blue to red, while the scale remains unchanged.

Once the color has transitioned after 500 milliseconds, then the scale transition begins, which lasts another 500 milliseconds. In total, the full transition takes 1 second.

The `transition` property is a shorthand property that combines several transition-related properties together. Listing 9-4 defines the same transitions as in Listing 9-3, but it uses separate transition properties.

Listing 9-4. Using the separate transition properties

```
<style>
  button.fancy-button {
    background: blue;
    transition-property: background-color, transform;
    transition-duration: 500ms, 500ms;
    transition-delay: 0ms, 500ms;
  }

  button.fancy-button:hover {
    background: red;
    transform: scale(1.1);
  }
</style>

<button class="fancy-button">Fancy Button</button>
```

Transition Time Units

The timing for transitions (and animations, as we'll see in the next section) can be specified in either seconds (`s`) or milliseconds (`ms`). Fractional values can be used; that is, 500 milliseconds can be specified as `500ms` or `0.5s`.

Animating an Element's Initial State with `@starting-style`

So far, we've seen how to transition an element's styles between two states, defined in two separate CSS rules. You can also animate an element's initial appearance on the page with the `@starting-style` at-rule.

Consider the fancy button from the previous examples. Let's suppose you want the button to fade in to the page. You can add an opacity transition to the button and set its starting opacity to 0. Listing 9-5 shows this.

Listing 9-5. Transitioning with `@starting-style`

```
<style>
  button.fancy-button {
    background: blue;
    opacity: 1;
    transition: opacity 500ms;

    @starting-style {
      opacity: 0;
    }
  }
</style>

<button class="fancy-button">Fancy Button</button>
```

When the page first loads, the button is invisible because its opacity is set to `0` in the `@starting-style` block. The element itself has an opacity of 1 and an opacity transition. Once the starting style is applied, the 500ms transition immediately runs.

`@starting-style` is a newer CSS feature, and at the time of writing, it's well supported in the latest versions of modern browsers. If you need to target older browsers, this technique may not work.

Easing Functions

It's hard to notice with a short duration of 500 milliseconds, but the transitions from Listings 9-4 and 9-5 don't happen at a linear rate. That is, they start out slow, speed up in the middle, then slow down again at the end. You can see this more clearly if you change the transition duration to a longer value like five seconds.

This is the default transition timing function, which is called `ease`. This function specifies the rate at which the animated transition is applied. There are other built-in timing functions as well. Formally, these are specified as easing functions. These functions are visualized by plotting a graph, with the time on the X-axis and the transition progress on the Y-axis.

These functions are usually given as *Cubic Bezier* curves. A Cubic Bezier curve is created by specifying four points on a graph. The points are plotted, and a curve is drawn. A good way to think of how the curve is drawn is like this: Think of a straight line drawn between the first and last points. Then, the second and third points "bend" the line up and down toward them to make a curve.

In a Cubic Bezier curve for a CSS easing function, the first point is always (0, 0) and represents the start state of the animation or transition. The last point is always (1, 1) and represents the end state. The X-coordinate of the two middle points must be between 0 and 1, or else it is not considered a valid easing function and will be ignored.

Since the first and last points of the curve are always (0, 0) and (1, 1), we only need to specify the X and Y coordinates of the two middle points that "bend" the curve:

```
transition-timing-function:
  cubic-bezier(.17, .67, .9, .6);
```

This `cubic-bezier` function yields the curve shown in Figure 9-1.

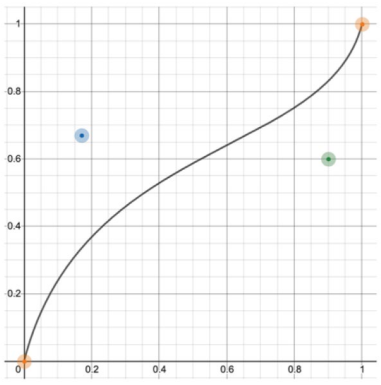
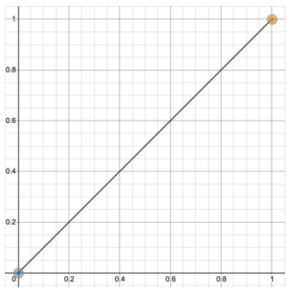
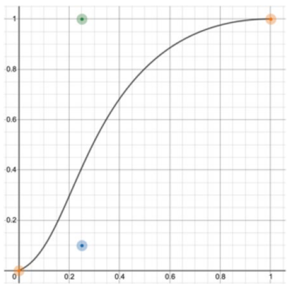
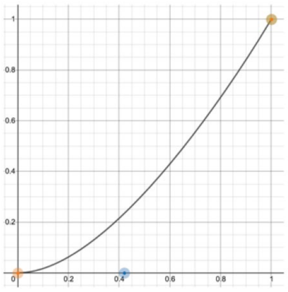
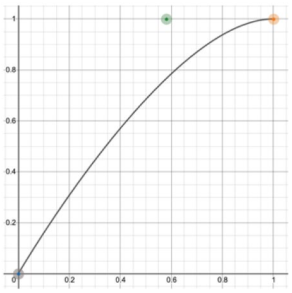


Figure 9-1. *The plotted Bezier curve*

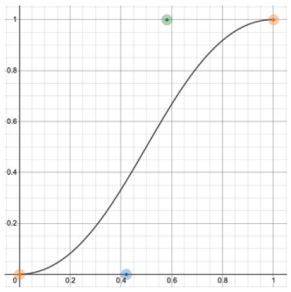
There are several built-in easing functions which you can specify by name instead of having to use a `cubic-bezier` function expression. These functions are shown in Table 9-1 with Figures 9-2 through 9-4.

Table 9-1. *Built-in easing functions*

Function	Equivalent `cubic-bezier` Values	Graph
`linear`	0.0, 0.0, 1.0, 1.0	
`ease`	0.25, 0.1, 0.25, 1.0	
`ease-in`	0.42, 0.0, 1.0, 1.0	
`ease-out`	0.0, 0.0, 0.58, 1.0	

(*continued*)

Table 9-1. (*continued*)

Function	Equivalent `cubic-bezier` Values	Graph
`ease-in-out`	0.42, 0.0, 0.58, 1.0	

The animation progress (the Y-axis) can also bend above 1 or below 0 to create a "bouncing" effect for some properties. For example, the easing function `cubic-bezier(0.680, -0.550, 0.265, 1.550)` results in the graph shown in Figure 9-2.

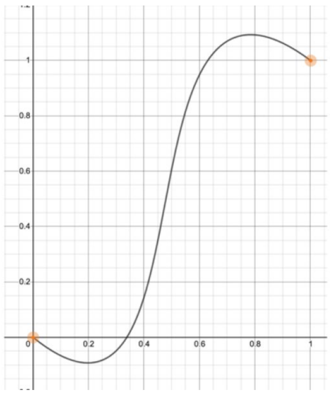


Figure 9-2. *An easing function with a "bouncing" effect*

Notice how the high and low points cause the curve to bend above 1 and below 0. This means, for the earlier example, that the `scale` transform would drop below 1 (the initial value) and go above 1.1 (the end value).

It can be difficult to craft these functions by hand, so there are several great resources online to help you design custom ones:

- Ceaser by Matthew Lein

  `https://matthewlein.com/tools/ceaser`

- cubic-bezier.com by Lea Verou

  `https://cubic-bezier.com`

Easing functions can also be specified as *step functions*. A step function divides the transition into equally sized steps in a given direction. Instead of a smooth transition like with a Bezier curve, they jump from step to step, skipping the intermediate states.

These are specified as `steps(step-count, direction)`. `step-count` is a number indicating the number of these steps, and `direction` is one of the following values:

- `jump-start`, `start`: The change in state happens at the beginning of each step, beginning with the start of the transition. Because the first "jump" happens immediately, the initial state of the transition is effectively lost.
- `jump-end`, `end`: The change in state happens at the end of each step, ending with the end of the transition. With this value, the end state of the transition is lost.
- `jump-none`: With this value, the start and end state of the transition are both preserved. The first step is the initial state, and the last step is the end state.
- `jump-both`: With this value, the start and end state of the transition are both lost.

CSS transitions can be a powerful tool to add better interactivity to a website or app. However, they are limited in transitioning from one initial state to one final state. With CSS animations, which we'll look at next, we can have an arbitrary number of states to create even more interesting effects.

Animations

While transitions provide an animated transition from a start state to an end state, CSS animations can animate between any arbitrary number of states.

Like transitions, animations are specified by the properties that change. Instead of being specified in a property like `transition`, they are specified in special at-rules: `@keyframes`. The `@keyframes` rule defines the various CSS properties to be applied at given steps during the animation, and the browser will automatically animate between these states.

A `@keyframes` rule is given an identifier and contains two or more blocks of CSS properties. Each block is labeled with a percentage representing a fraction of the total animation duration. In the example from Listing 9-6, the element's background color will change from red to blue to green.

Listing 9-6. A basic CSS animation

```
@keyframes colors {
  0% {
    background: red;
  }

  50% {
    background: blue;
  }

  100% {
    background: green;
  }
}
```

Here, we are defining a `@keyframes` rule named `colors`. The initial state of an element using this animation will have a background color of red. At the halfway point of the animation, it will have a color of blue. Finally, at the end state, it will have a color of green. Like with transitions, the browser will automatically calculate all the intermediate colors for the animation.

`0%` can be replaced with the keyword `from`, and `100%` can be replaced with the keyword `to`, but this is optional and has the same meaning.

Once an animation is defined in a `@keyframes` rule, we must apply it to an element. The name of the `@keyframes` rule is referenced in the `animation` property of a CSS rule matching the element to be animated. Listing 9-7 shows an example of applying the animation from Listing 9-6 to an element.

Listing 9-7. Applying the animation to an element

```
<style>
  @keyframes colors {
    0% {
      background: red;
    }

    50% {
      background: blue;
    }

    100% {
      background: green;
    }
  }

  .animated-box {
    animation: colors 2s;
    width: 10rem;
    height: 10rem;
  }
</style>

<div class="animated-box"></div>
```

The `animation` property can take many forms, as it is a shorthand for several other animation-related properties. Here are some example usages of the `animation` property:

- `animation: colors 2s ease-in-out 4s both;`
  - This is shorthand for
    - `animation-name: colors;`
    - `animation-duration: 2s;`
    - `animation-timing-function: ease-in-out;`
    - `animation-delay: 4s;`
    - `animation-fill-mode: both;`

- `animation: colors 2s linear infinite;`
    - This is shorthand for:
        - `animation-name: colors;`
        - `animation-duration: 2s;`
        - `animation-timing-function: linear;`
        - `animation-iteration-count: infinite;`

Basic Animation Properties

The most commonly used animation properties include

- `animation-name`: The name of the animation to use. There must be a matching `@keyframes` rule.
- `animation-duration`: The total duration of the animation. This can be specified in seconds (`s`) or milliseconds (`ms`).
- `animation-timing-function`: The easing function to use for the animation. See the previous section on easing functions for examples of these functions.

Delaying the Start of the Animation

By default, an animation will start immediately. The `animation-delay` property will delay the start of the animation by the given time. Like animation-duration, this can be specified in seconds or milliseconds.

You can also give a negative value for `animation-delay`. If you do this, the animation will start immediately at the given point of elapsed time. For example, if an animation has a duration of `1s`, and the delay is `-500ms`, the animation will start immediately at the 500-millisecond mark.

The Animation Fill Mode

If you try the example in Listing 9-7, you might be surprised by the behavior. The element indeed animates, starting from red and going through blue into green over two seconds, but then the element seems to disappear. It's still there, but its background color has changed back to the default setting of transparent.

Once an animation finishes, any styles changed during the animation will by default revert back to the styles applied before the animation. In this case, we don't give a background color for the element outside of the animation, so it reverts back to transparent.

If we had modified the CSS rule so it also had a background of yellow, the animation would play and then the element would turn yellow.

This behavior also applies if the animation has a delay. If the colors animation has a delay, then the element will start out yellow until the animation runs. After the animation completes, it will go back to yellow.

You can change this behavior with the `animation-fill-mode` property. It defines how properties from the animation are applied to the element before the animation starts, after the animation ends, or both. Before we go over the different values of this property, consider the example in Listing 9-8.

Listing 9-8. An example animation with a delay

```
<style>
  @keyframes color {
    from {
      background: red;
    }

    to {
      background: blue;
    }
  }

  .animate {
    background: yellow;
    animation: color 2s;
    animation-delay: 2s;
    width: 10rem;
    height: 10rem;
  }
</style>

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

We have a box with a yellow background. It animates its background color starting from red and transitioning to blue. The animation lasts two seconds, and there is a two-second delay before it starts. Let's explore the different values of `animation-fill-mode` and what effect they would have on this animation.

If we set it to `none`, the element is yellow for two seconds. It then immediately switches to red, and over the next two seconds, it transitions to blue. After the animation completes, the background color immediately switches back to yellow. With none, the animation styles are only applied for the duration of the animation. This is the default behavior if no fill mode is specified.

If we set it to `forwards`, the element again starts out yellow and stays that way for two seconds. It immediately switches to red, and over the next two seconds, it transitions to blue. After the animation completes, the background color remains blue. With `forwards`, the styles from the last keyframe of the animation remain applied to the element after the animation ends.

If we set it to `backwards`, the element starts out red, which it remains for the next two seconds. Then the animation runs as expected, transitioning to blue. Once the animation is done, the background color immediately switches back to yellow. With `backwards`, the styles from the first keyframe are applied during the animation delay period.

Finally, if we want the behavior of both `forwards` and `backwards`, we can set it to `both`. The element starts out red, stays that way for two seconds, then transitions to blue. Once the animation finishes, the element remains blue. With `both`, the styles from the first keyframe are applied during the delay, and the styles from the last keyframe are applied after the animation finishes.

Running an Animation Multiple Times

By default, an animation runs only once. You can run the animation more than once by using the `animation-iteration-count` property. This can be set to a number specifying how many iterations, or you can have it loop forever by specifying the keyword `infinite`.

The iteration count doesn't have to be an integer value. For example, you can specify an `animation-iteration-count` of `0.5` and the animation will play once, but only to its halfway point.

Running an Animation Backward

You can control which direction an animation plays in by using the `animation-direction` property. This can be `normal` (the default), which plays the animation forward, or `reverse`, which plays the animation backward.

You can also use the values `alternate` (runs the animation forward first, then backward) or `alternate-reverse` (runs the animation backward first, then forward).

Pausing and Resuming an Animation

You can control whether or not the animation is currently playing by setting the `animation-play-state` property. This could be manipulated with JavaScript to pause and resume an animation, for example. Accepted values are `running` and `paused`.

When the animation is changed from `running` to `paused`, and later changed back to `running`, the animation will continue from where it left off – it won't start over from the beginning.

Applying Multiple Animations

You can apply more than one animation to an element. The `animation` shorthand property, as well as all the other animation properties, supports a comma-separated list of multiple animations. The example in Listing 9-9 applies color and rotation animations independently.

Listing 9-9. Applying multiple animations to an element

```
<style>
  @keyframes color {
    from {
      background-color: red;
    }
    to {
      background-color: blue;
    }
  }

  @keyframes spin {
    from {
      transform: rotate(0);
    }
    to {
      transform: rotate(360deg);
    }
  }

  .animate {
    width: 10rem;
    height: 10rem;
    animation: color 5s alternate infinite,
               spin 1s linear infinite;
  }
</style>

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

This element's color will cycle infinitely between blue and red every five seconds while also spinning at a rate of one rotation per second at the same time, as shown in Figure 9-3.

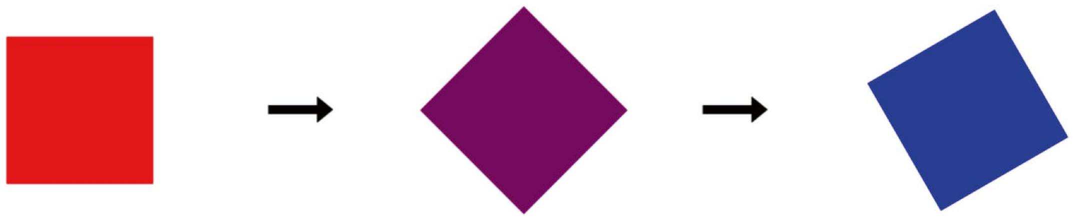


Figure 9-3. *The multiple animations being applied*

Applying Multiple Animations That Change the Same Property

If you have more than one animation that works on the same CSS property, by default you won't see both animations – just the last one you applied. Listing 9-10 attempts to apply both a translate and rotate animation using multiple keyframe animations.

Listing 9-10. Attempting to animate multiple transforms

```
<style>
  @keyframes move {
    from {
      transform: translate(0);
    }

    to {
      transform: translate(200px);
    }
  }

  @keyframes spin {
    from {
      transform: rotate(0deg);
    }

    to {
      transform: rotate(360deg);
    }
  }

  .animated-box {
    animation: move 5s, spin 5s;
    animation-fill-mode: both;
    background: skyblue;
    width: 150px;
    height: 150px;
  }
</style>

<div class="animated-box"></div>
```

If you try this example, though, you'll see that the element rotates but does not translate. This is because the `spin` animation "cancels out" the `transform` property from the `move` animation.

We can fix this by using the `animation-composition` property, which tells the browser how to handle multiple animations that change the same property. The default value of this property is `replace`, which causes the behavior we're seeing with Listing 9-10.

If you instead set animation-composition to `add`, you'll see the two transforms are combined. The element translates and rotates as expected.

Performance Implications

Transitions and animations are powerful. But with great power comes great responsibility. Overusing them, or using them for certain expensive properties, can result in poor performance of your page.

Property Types

There are a few different ways we can categorize CSS properties. There are properties for layout, paint, and composite operations.

Layout Properties

Layout properties are properties that affect how an element and its surrounding content are laid out on the page. This category includes properties like `width`, `height`, `padding`, and `margin`. These are generally the most expensive to animate.

If you animate the margin of an element, for every frame of the animation, the size taken up by the element changes. This will affect the layout of surrounding elements. When those elements' layouts are adjusted, that could trigger even more elements to recalculate their layout. This chain reaction is known as layout thrashing and can be very costly for performance.

Paint Properties

Paint properties affect how an element is painted on the screen, such as `color` or `background-image`. These properties don't affect layout, so they aren't as expensive as layout properties. However, if overused, they can still cause performance issues, particularly on mobile devices.

Composite Properties

Composite properties are properties such as transform and opacity. These properties don't affect layout and are much cheaper than the other property types. Additionally, the device's Graphics Processing Unit (GPU) can also assist with these animations, which takes a load off the CPU and makes the animations smoother.

Giving a Hint As to What Properties Will Be Animated

If you're having animation performance problems, as a last resort you can try the `will-change` property. Setting this on an element gives a hint to the browser that the specified properties will be changing due to an animation or transition. Listing 9-11 has an example of using this property.

Listing 9-11. Example usage of the `will-change` property

```
.my-element {
  will-change: transform, opacity;
}
```

This proves a hint that the transform and opacity properties will be changing, so the browser can take that into account when optimizing for the animation.

As mentioned, this is a last resort and should be used sparingly. The browser already does a good job optimizing layout, paint, and composite operations to help keep your animations and transitions smooth. Overusing `will-change`, or using it prematurely, can interfere with that optimization and could actually make performance worse.

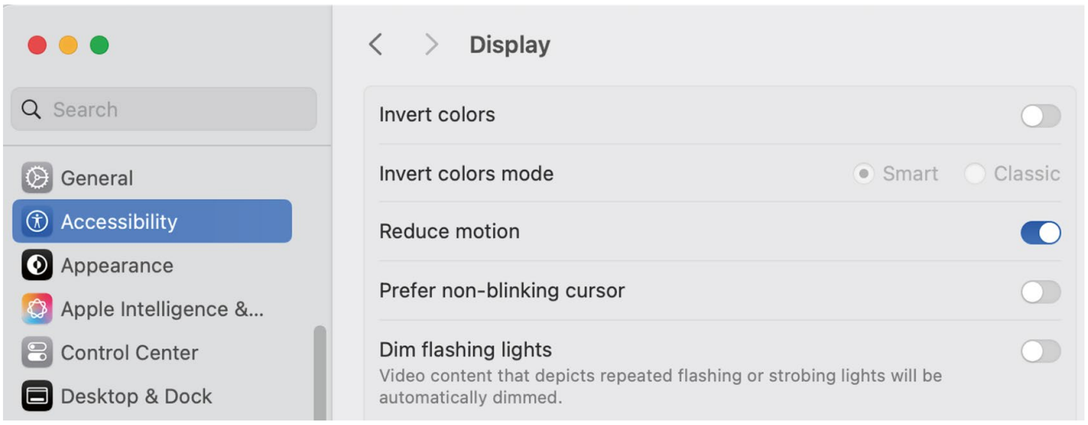
Avoiding Simultaneous Animations

Try to limit the number of animations running simultaneously. Even efficient animations can cause performance issues when combined with several others at once, particularly if one or more of them already have performance issues, such as animating a layout property.

When you do perform multiple animations at once, it can be useful to test each in isolation to get an idea of which ones will contribute to the most performance problems.

Considering Accessibility

Some users may have vestibular or seizure disorders that can be triggered by rapidly moving or flashing elements in your pages. You should be mindful of this when designing your animations. Most modern operating systems allow users to disable, or reduce, animations to help alleviate this. Figure 9-4 shows this option in the macOS accessibility settings ("Reduce motion").



Invert colors ○
Invert colors mode [✓ Smart] Classic
Reduce motion ●
Prefer non-blinking cursor ○
Dim flashing lights ○
Video content that depicts repeated flashing or strobing lights will be automatically dimmed.

Figure 9-4. *The accessibility options in macOS*

By default, even if a user has disabled animations or reduced motion, your CSS animations and transitions will still run. You can, and should, detect this setting by using the `prefers-reduced-motion` media query and adjusting or disabling your animations accordingly. We'll get more into media queries in Chapter 12, but here's how to use them. Consider Listing 9-12, which has a basic animated loading spinner.

Caution If you are sensitive to fast motion or animation, I recommend that you don't run the code for Listing 9-12 as it is a fast animation that could be triggering.

Listing 9-12. A spinning element

```
<style>
  @keyframes spin {
    from {
      transform: rotate(0deg);
    }
    to {
      transform: rotate(360deg);
    }
  }

  .loader {
    width: 10rem;
    height: 10rem;
    background: skyblue;
    animation: spin 500ms linear infinite;
  }
</style>

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

This results in a square that spins very quickly, making one full rotation every 500ms. This could trigger seizures or other issues, as it moves very fast. We can conditionally disable the animation by using the `prefers-reduced-motion` media query, as shown in Listing 9-13.

Listing 9-13. Using the prefers-reduced-motion media query

```
@media (prefers-reduced-motion: reduce) {
  .loader {
    animation: none;
  }
}
```

When this code runs on a device where the user has reduced motion, the box will not be animated. The prefers-reduced-motion media query has two supported values: `reduce` and `no-preference`.

Note that you don't necessarily have to disable the animation altogether. For users that prefer reduced motion, you could use a more subtle animation, like a slow fade animation, that will be less triggering.

Scroll-Driven Animations

Traditional animations are managed by the *document timeline*. This is a timeline based on elapsed time since the page loaded. Time starts at zero and moves forward in a linear fashion, and animations play as this time passes (though, depending on the easing function, the animation itself may not be linear).

There's another type of timeline that can be used for an animation: the *scroll progress timeline*. This timeline changes as you scroll up and down. While scrolling down, the animation runs normally. If you scroll back up, the animation runs in reverse.

The animation timeline is specified with the `animation-timeline` property. Listing 9-14 has an example of a scroll-driven animation. It's a progress bar animation that grows and shrinks as the page scrolls up and down.

Listing 9-14. Creating a scroll-driven animation

```
@keyframes progress {
  from {
    transform: scaleX(0);
  }

  to {
    transform: scaleX(1);
  }
}

.progress-bar {
  animation: progress linear;
  animation-timeline: scroll();
  background: blue;
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 5px;
  transform-origin: left;
}
```

The interesting part here is the use of the `animation-timeline` property. It uses the `scroll` function which, by default, uses the root element as the scroll container. The result is a blue progress bar fixed to the top of the page. It uses the `scaleX` transform with the origin at the left edge to have the effect of a growing progress bar animation.

At the time of writing, the animation-timeline property is not yet fully supported in all major browsers. As of May 2025, only Chrome and Edge support this property. More browsers will support this feature in the future.

Summary

- A transition is used to animate an element between two states, while an animation can have any number of states.
- An easing function determines the timing of the animation progress.
- Transitions and animations have a duration and an optional delay.
- Animations have the `animation-fill-mode` property, which determines how styles are applied before and after the animation is performed.
- Try to avoid animating layout properties, as these can negatively affect performance.
- Use the `prefers-reduced-motion` media query to improve the accessibility of your animations.
- An animation has a timeline. By default, it's the document timeline which follows the elapsed time since the page was loaded.
- You can also use a scroll timeline to create scroll-driven animations that change as the user scrolls.

CHAPTER 10

Flexbox

The flexible box layout model, more commonly known as *flexbox*, is a powerful tool for building layouts with CSS. It's not quite as powerful as CSS Grid, which we'll look at in Chapter 11, but it can solve many layout problems.

Flexbox is a one-dimensional layout that can position elements either horizontally or vertically. An element using flexbox as its layout is referred to as a *flex container*, and the elements inside it are *flex items*.

Basic Concepts

Let's go over the basic concepts to understand and use flexbox layouts.

Creating a Flex Container

To create a flex container, set an element's `display` property to `flex`. This makes the element a block element with a flexbox layout. If you need the behavior of an inline element, but still want to use flexbox, you can set `display` to `inline-flex`.

Direction

A flex container has a direction, defined by the `flex-direction` property. The meaning of this property depends on whether the system is using a left-to-right (LTR) language or a right-to-left (RTL) language. The explanation of flex direction values here assumes an LTR language; for an RTL language, they are reversed.

The `flex-direction` property supports these four values:

- `row`: Flex items are laid out horizontally, from left to right. This is the default behavior if `flex-direction` is not specified.
- `row-reverse`: Flex items are laid out horizontally, from right to left.

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_10

- `column`: Flex items are laid out vertically, from top to bottom.
- `column-reverse`: Flex items are laid out vertically, from bottom to top.

These `flex-direction` values are visualized in Figure 10-1.

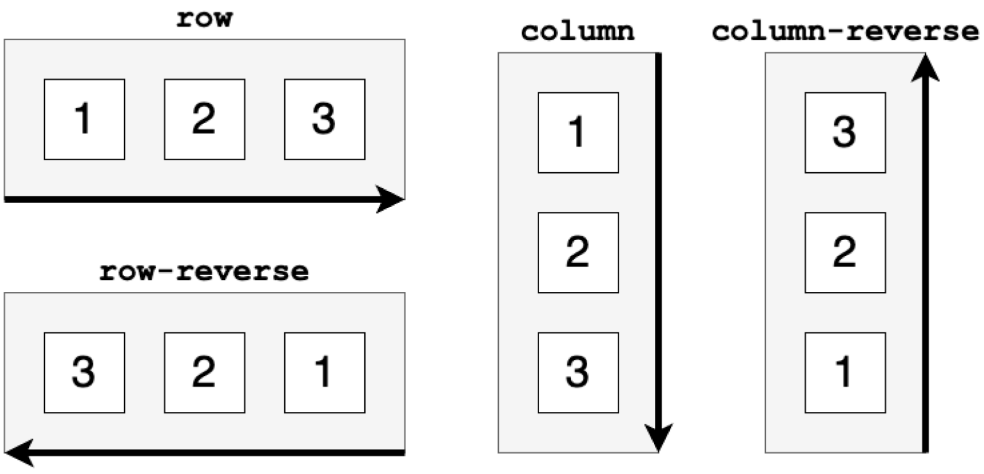


***Figure 10-1.** The different values for the `flex-direction` property*

Axis

Related to the direction is the concept of the *axis*. A flex container has two axes: the *main axis*, which runs along the direction specified in `flex-direction`, and the *cross axis*, which runs perpendicular to it. Figure 10-2 shows the main and cross axes for both horizontal (`row`) and vertical (`column`) flex layouts.

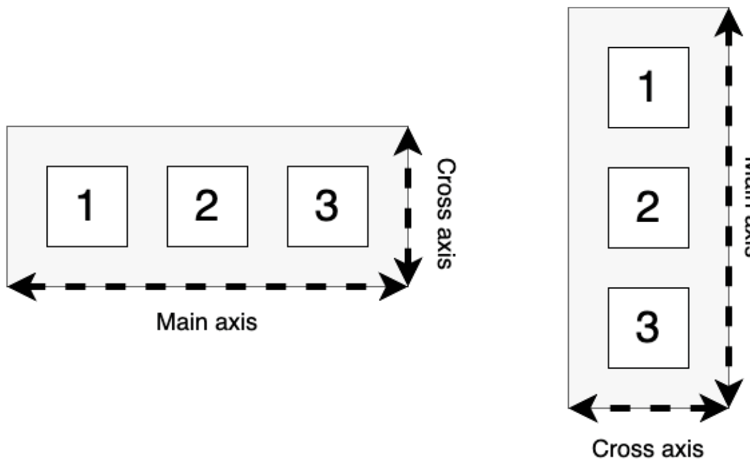


Figure 10-2. *The main axis and cross axis*

A Basic Flexbox Layout

Let's create our first flexbox layout. Listing 10-1 lays out some elements in a horizontal flex container.

Listing 10-1. A basic flexbox layout

```
<style>
  .container {
    background: #e2e8f0;
    display: flex;
    flex-direction: row;
  }

  .item {
    padding: 16px;
    background: #bfdbfe;
    border: 1px solid black;
  }
</style>
```

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

The resulting layout is shown in Figure 10-3.

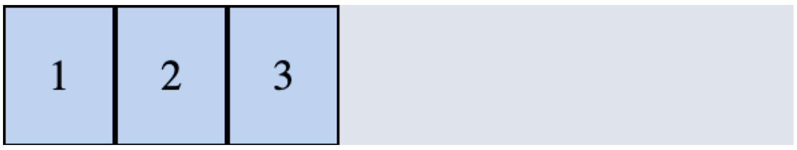


Figure 10-3. *The rendered flexbox layout*

Adding Space Between Items

The flex items are arranged horizontally, with no space between them. You can add space between flex items by using the gap property. Listing 10-2 adapts the previous flexbox layout and adds a gap.

Listing 10-2. A flexbox layout with a gap

```
<style>
  .container {
    background: #e2e8f0;
    display: flex;
    flex-direction: row;
    gap: 16px;
  }

  .item {
    padding: 16px;
    background: #bfdbfe;
    border: 1px solid black;
  }
</style>
```

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

As Figure 10-4 shows, there is now 16 pixels of spacing between the flex items.

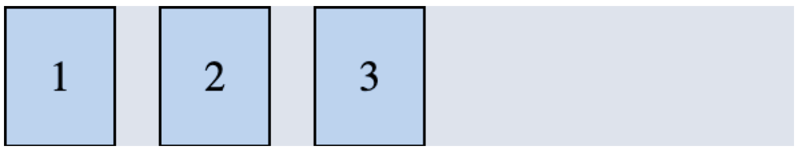


Figure 10-4. *The rendered layout with a gap between elements*

Sizing of Flex Items

What happens if the flex items don't fit into the flex container? Listing 10-3 has a flexbox layout with a constrained width and items that, given their widths, shouldn't be able to fit inside of it.

Listing 10-3. A flex container that isn't wide enough to fit its items

```
<style>
  .container {
    display: flex;
    background: #e2e8f0;
    width: 500px;
    gap: 16px;
    padding: 16px;
    border: 1px solid #000000;
  }

  .item {
    width: 300px;
    text-align: center;
    padding: 16px;
    background: #bfdbfe;
    border: 1px solid #000000;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

Figure 10-5 shows the resulting layout.

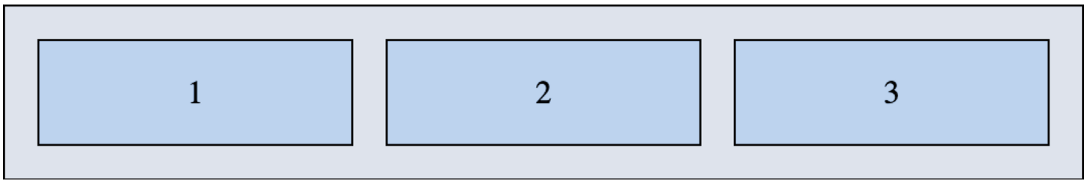


Figure 10-5. *The rendered layout*

The container has a width of 500 pixels and contains three items, each with a width of 300 pixels. Yet, the items fit neatly inside the container and don't overflow it. If you were to inspect these elements with the browser's developer tools, you'd see that they have an actual width of 156 pixels.

This shrinking behavior is a feature of flexbox. Items will shrink to fit within the container if possible. This is controlled by the `flex-shrink` property, which we'll look at later in this chapter.

Sometimes, even with `flex-shrink`, elements can't be made small enough to fit within the container. When this happens, the items will shrink as much as possible and then will overflow the container. Listing 10-4 has an example of this.

Listing 10-4. Items that can't shrink enough to fit inside the flex container

```
<style>
  .container {
    display: flex;
    background: #e2e8f0;
    width: 100px;
    gap: 16px;
    padding: 16px;
    border: 1px solid #000000;
  }

  .item {
    width: 300px;
    text-align: center;
    padding: 16px;
    background: #bfdbfe;
    border: 1px solid #000000;
  }
</style>

<div class="container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
</div>
```

Figure 10-6 shows the result.

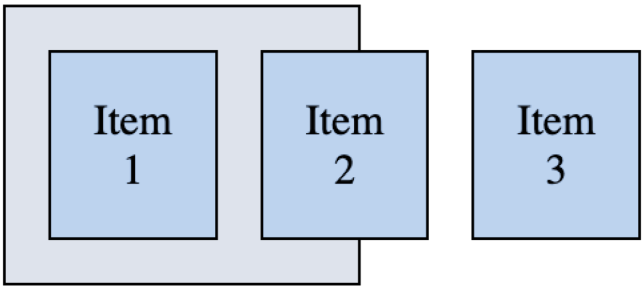


Figure 10-6. *Flex items overflowing the container*

The flex items have shrunk as much as they can be, and then they overflow the container due to its constrained width of 100 pixels.

Wrapping Flex Items

One solution to the problem in Figure 10-6 is to allow flex items to wrap over multiple lines. This increases the container's height but prevents items from overflowing. Listing 10-5 shows how we can use the `flex-wrap` property to do this.

Listing 10-5. Wrapping a layout with `flex-wrap`

```
<style>
  .container {
    display: flex;
    flex-wrap: wrap;
    background: #e2e8f0;
    width: 100px;
    gap: 16px;
    padding: 16px;
    border: 1px solid #000000;
  }

  .item {
    width: 300px;
    text-align: center;
    padding: 16px;
    background: #bfdbfe;
    border: 1px solid #000000;
  }
</style>

<div class="container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
</div>
```

This causes the items to wrap across multiple lines, as Figure 10-7 shows.

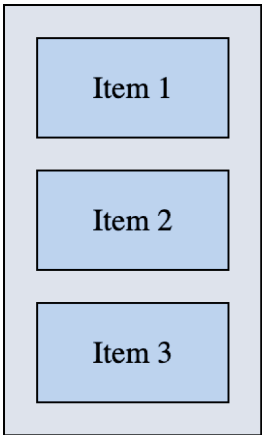


Figure 10-7. *The wrapped flex items*

Growing Flex Items to Fill Available Space

By default, if the flex items are not large enough to fill the container, there will be empty space after the flex items. Consider the layout in Listing 10-6.

Listing 10-6. Empty space in the container

```
<style>
  .container {
    background: #e2e8f0;
    display: flex;
    flex-direction: row;
    gap: 16px;
    width: 400px;
    padding: 16px;
    border: 1px solid black;
  }

  .item {
    padding: 16px;
    background: #bfdbfe;
    border: 1px solid black;
  }
</style>
```

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

As Figure 10-8 shows, there is empty space left over in the container.

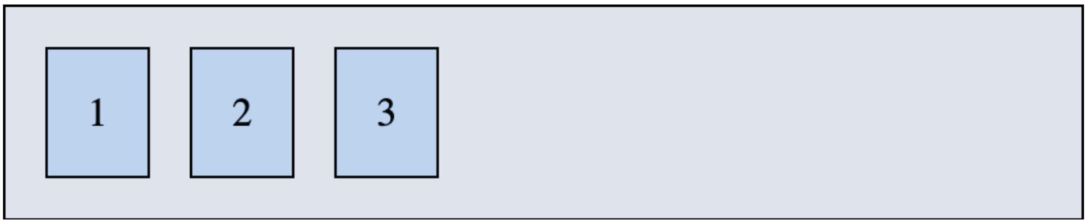


Figure 10-8. *Empty space in the flex layout*

This behavior can be changed using the `flex-grow` property, which allows flex items to grow to fill available space. This property is set on the flex items rather than the flex container. By default, `flex-grow` is `0`, which means the items don't grow at all.

A `flex-grow` value is a number that specifies the item's growth relative to the other items. If `flex-grow` is set to the same value for all items, they'll all grow equally to fit the container, like Listing 10-7 shows.

Listing 10-7. Adding an equal `flex-grow` property

```
<style>
  .container {
    background: #e2e8f0;
    display: flex;
    flex-direction: row;
    gap: 16px;
    width: 400px;
    padding: 16px;
    border: 1px solid black;
  }
  .item {
    padding: 16px;
    background: #bfdbfe;
    border: 1px solid black;
    text-align: center;
    flex-grow: 1;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

As Figure 10-9 shows, the items all grow equally to fit the container.

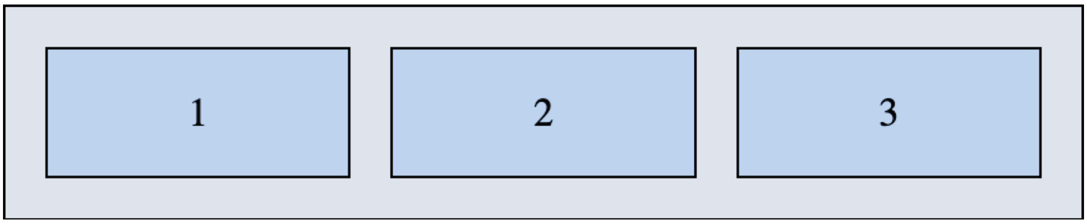


Figure 10-9. *The flex items grow to fit the container*

You can customize this behavior by specifying different `flex-grow` values for different flex items. Suppose we want item 2 to grow twice as much as items 1 and 3. We can specify different relative `flex-grow` values, and they will be sized accordingly, as Listing 10-8 shows.

Listing 10-8. Specifying different `flex-grow` values

```
<style>
  .container {
    background: #e2e8f0;
    display: flex;
    flex-direction: row;
    gap: 16px;
    width: 500px;
    padding: 16px;
    border: 1px solid black;
  }

  .item {
    width: 100px;
    padding: 16px 0;
    background: #bfdbfe;
    border: 1px solid black;
    text-align: center;
    flex-grow: 1;
  }

  .item2 {
    flex-grow: 2;
  }
</style>

<div class="container">
  <div class="item item1">1</div>
  <div class="item item2">2</div>
  <div class="item item3">3</div>
</div>
```

Figure 10-10 shows the resulting layout.

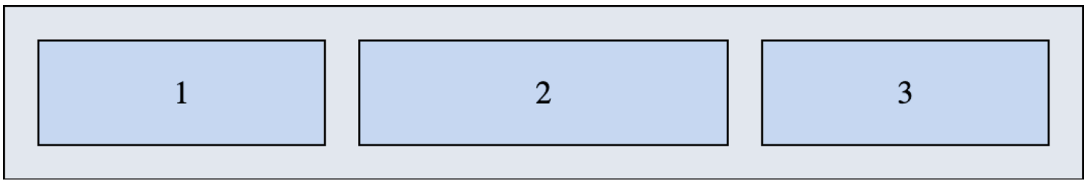


Figure 10-10. *The layout with `flex-grow` values applied*

Item 2 has twice the `flex-grow` as items 1 and 3. This doesn't mean that item 2 is twice as wide as the others, but rather that the amount it grows by is twice that of the others. Items 1 and 3 grow by about 42 pixels, while item 2 grows by about 84 pixels. The exact amount may vary slightly due to sub-pixel rendering of the items.

Shrinking Flex Items to Fit the Available Space

We saw earlier that if the flex items exceed the size of the container, the browser will try to shrink them to fit. By default, it will try to shrink all elements evenly. You can change this behavior with the `flex-shrink` property.

Just as `flex-grow` controls the relative amount by which a flex item grows, `flex-shrink` controls the relative amount by which a flex item shrinks. By default, `flex-shrink` is `1`, which is why the items shrink evenly. Listing 10-9 has three flex items that are too big to fit the container. The middle item has a `flex-shrink` of 2.

Listing 10-9. Specifying a `flex-shrink` of 2

```
<style>
    .container {
      background: #e2e8f0;
      display: flex;
      flex-direction: row;
      gap: 16px;
      width: 500px;
      padding: 16px;
      border: 1px solid black;
    }

    .item {
      width: 400px;
      padding: 16px 0;
      background: #bfdbfe;
      border: 1px solid black;
      text-align: center;
    }

    .item2 {
      flex-shrink: 2;
    }
  </style>
```

```
<div class="container">
  <div class="item item1">1</div>
  <div class="item item2">2</div>
  <div class="item item3">3</div>
</div>
```

The container is 500 pixels wide, and each item is 400 pixels wide, so they must shrink to fit inside the container. Item 2 has a `flex-shrink` of 2, which means it will shrink twice as much as the other items. Figure 10-11 shows the resulting flex items.

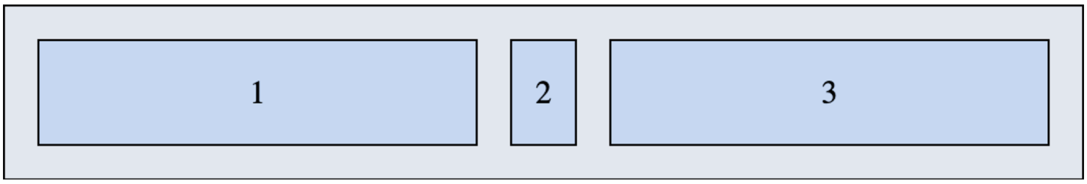


Figure 10-11. *The middle item shrinks much more than the others*

The total width taken up by all three items would be 1,200 pixels. The items must shrink to fit the 500-pixel container. Items 1 and 3 are shrunk by approximately 183 pixels, and item 2 is shrunk by approximately 367 pixels, which is about twice that of the others. Again, this is not exact due to sub-pixel rendering.

Setting the Initial Size of a Flex Item

The `flex-basis` property sets the initial size of a flex item along the main axis before the flex-grow and `flex-shrink` factors are applied. The meaning of flex-basis depends on the flex-direction of the container. For horizontal (`row`) flex containers, `flex-basis` will be the item's width. For vertical (`column`) containers, `flex-basis` will be the item's height.

If both `height`/`width` and `flex-basis` are set for an element in a flex layout, the `flex-basis` has higher precedence and will be used over the `height`/`width` value.

Alignment and Spacing

So far, we've seen how to size the flex items in a flexbox layout. In this section, we'll look at alignment (what happens when all the items are not the same size in the cross axis) and spacing (how leftover space is distributed).

The Writing Mode

Some of the values for these properties depend on the *writing mode*. The writing mode is defined by the `writing-mode` property. This property determines the way block elements are laid out and how inline elements flow inside them.

Furthermore, the way the writing mode behaves depends on the user's language. In some languages, text flows from left to right (LTR); in others, it flows from right to left (RTL).

The default value is `horizontal-tb`. For LTR languages, elements flow from left to right; for RTL languages, elements flow from right to left. Block elements and lines of text flow from top to bottom.

Other values include `vertical-rl` and `vertical-lr`. For these values, elements flow vertically from top to bottom (for LTR languages) or bottom to top (for RTL languages). With `vertical-rl`, block elements and lines of text flow from right to left (for LTR languages) or left to right (for RTL languages). Finally, with `vertical-lr`, block elements and lines of text flow from left to right (for LTR languages) or right to left (for RTL languages).

Controlling Spacing on the Main Axis

If the container is wider than its items, the `justify-content` property determines how the remaining space is distributed. This property is set on the flex container and has several accepted values, each with different behavior:

- `flex-start`: Lays out all the items next to each other at the beginning of the main axis. This is shown in Figure 10-12.

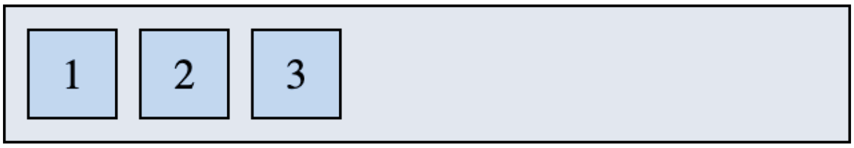


Figure 10-12. *The items arranged at the beginning of the main axis*

- `flex-end`: Lays out all the items next to each other at the end of the main axis. This is shown in Figure 10-13.

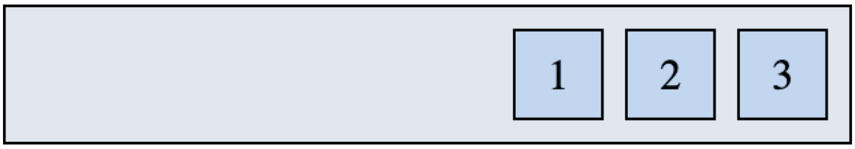


Figure 10-13. *The items arranged at the end of the main axis*

- `center`: Lays out all the items next to each other centered along the main axis. This is shown in Figure 10-14.

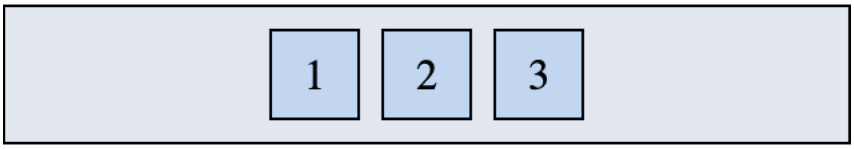


Figure 10-14. *The items centered along the main axis*

- `space-between`: Maximizes the space between the items. The first item is flush with the start of the main axis, the last item is flush with the end of the main axis, and the other items are distributed evenly. This is shown in Figure 10-15.

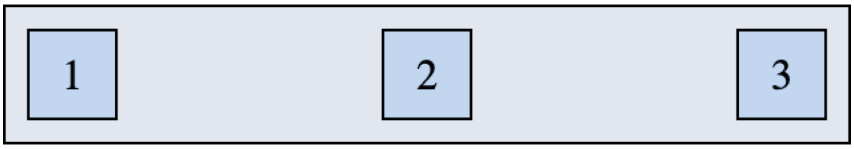


Figure 10-15. *Space is maximized between flex items*

- `space-around`: Like `space-between`, except there is also space at the beginning and end of the main axis. The space at the beginning and end of the main axis is half that of the other spaces. This is shown in Figure 10-16.

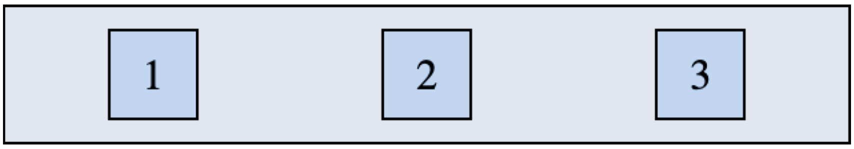


Figure 10-16. *Half spacing at the beginning and end of the main axis*

- `space-evenly`: Like `space-around`, except the space is even on all sides of all items. This is shown in Figure 10-17.

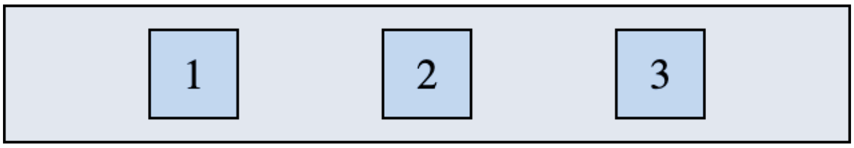


Figure 10-17. *All spacing is even*

Controlling Alignment on the Cross Axis

Also set on the container, the `align-items` property controls how flex items are aligned along the cross axis.

- `flex-start`: Items are aligned to the start of the cross axis. This is shown in Figure 10-18.

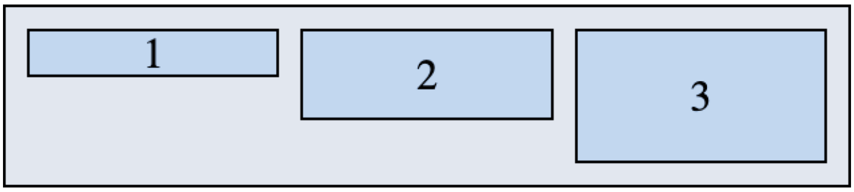


Figure 10-18. *Items are aligned to the start of the cross axis*

- `flex-end`: Items are aligned to the end of the cross axis. This is shown in Figure 10-19.

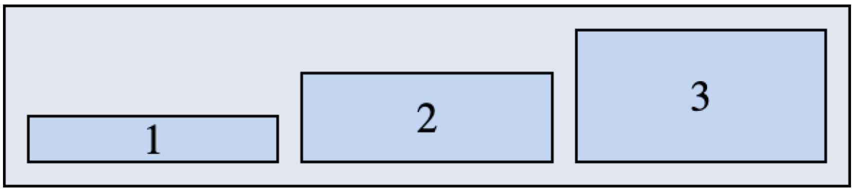


Figure 10-19. *Items are aligned to the end of the cross axis*

- `center`: Items are aligned to the center of the cross axis. This is shown in Figure 10-20.

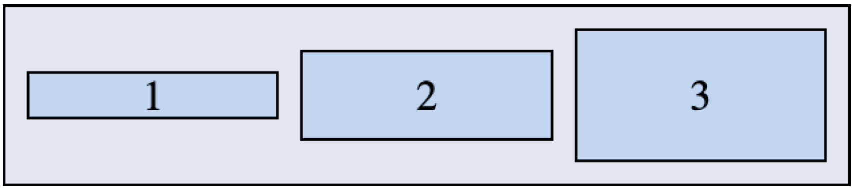


Figure 10-20. *Items are aligned to the center of the cross axis*

- `baseline`: Items are aligned along the baseline of their text content. This is shown in Figure 10-21.

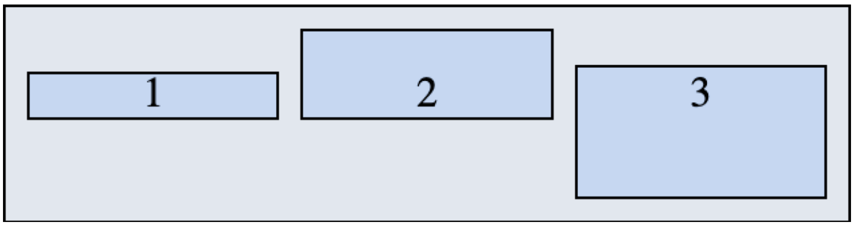


Figure 10-21. *Items aligned along the text baseline*

Controlling Spacing in the Cross Axis

When there are multiple rows (or columns in a column flexbox layout), and there is extra space in the cross axis, `align-content` specifies how this space is distributed.

- `flex-start`: Aligns the rows at the beginning of the cross axis. This is shown in Figure 10-22.

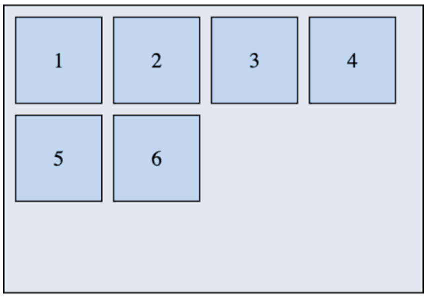


Figure 10-22. *Rows aligned at the beginning of the cross axis*

- `flex-end`: Aligns the rows at the end of the cross axis. This is shown in Figure 10-23.

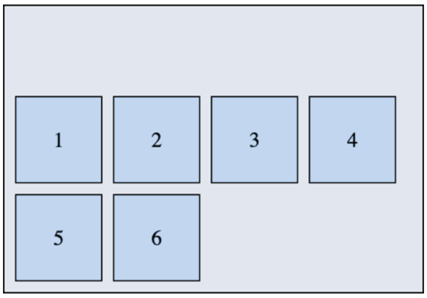


Figure 10-23. *Rows aligned at the end of the cross axis*

- `center`: Aligns the rows in the center of the cross axis. This is shown in Figure 10-24.

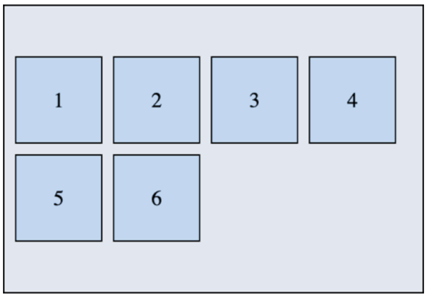


Figure 10-24. *Rows aligned in the center of the cross axis*

- `space-between`: Maximizes the spacing between rows. This is shown in Figure 10-25.

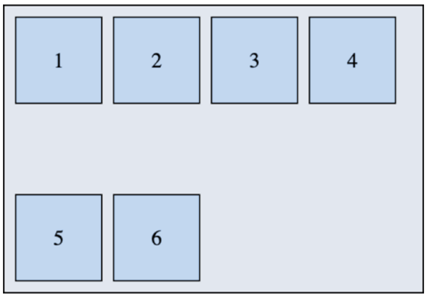


Figure 10-25. *Maximum space between the rows*

- `space-around`: Even spacing between the rows, with half space at the beginning and end of the cross axis. This is shown in Figure 10-26.

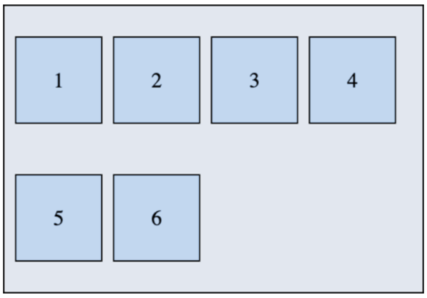


Figure 10-26. *Even spacing with half space at the beginning and end*

- `space-evenly`: Equal spacing around and between the rows. This is shown in Figure 10-27.

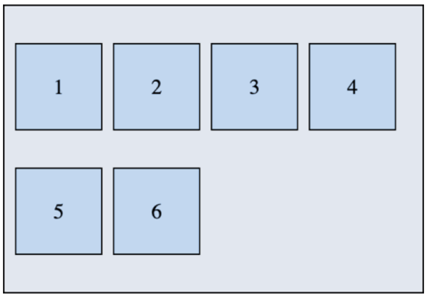


Figure 10-27. *Even spacing around and between*

Overriding Container Settings

By default, properties like `align-items` or `justify-content` apply to all flex items in the layout. However, you can override the behavior for a specific flex item by giving it the `align-self` or `justify-self` properties. Listing 10-10 has an example of using the `align-self` property.

Listing 10-10. Using the `align-self` property

```
<style>
  .container {
    background: #e2e8f0;
    display: flex;
    align-items: center;
    justify-content: center;
    gap: 8px;
    width: 200px;
    height: 100px;
    padding: 8px;
  }

  .item {
    padding: 16px;
    text-align: center;
    background: #bfdbfe;
    border: 1px solid black;
  }

  .item2 {
    align-self: flex-start;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item item2">2</div>
  <div class="item">3</div>
</div>
```

Figure 10-28 shows the resulting layout. Items 1 and 3 use the `align-items` value of `center`, but item2's `align-self` property overrides this, and it is aligned to the start of the cross axis.

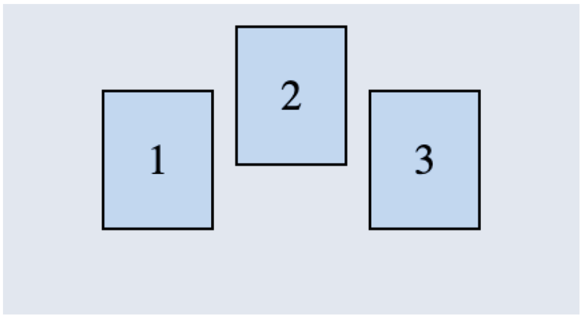


Figure 10-28. *An element with `align-self` set to `center`*

Changing the Flex Item Order

By default, flex items are laid out according to two factors:

- The order they occur in the HTML
- The value of the `flex-direction` property

This ordering can be changed by using the `order` property on flex items. This property, when set on a flex item, defines the order in which the item appears. Listing 10-11 shows how you can use the `order` property to change the order in which flex items appear.

Listing 10-11. Using the `order` property to change the order of flex items

```
<style>
  .container {
    background: #e2e8f0;
    display: flex;
    align-items: center;
    justify-content: center;
    gap: 16px;
    width: 200px;
    padding: 16px;
  }

  .item {
    padding: 16px;
    text-align: center;
    background: #bfdbfe;
    border: 1px solid black;
  }

  .item2 {
    order: 3;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item item2">2</div>
  <div class="item">3</div>
</div>
```

Figure 10-29 shows the resulting layout. While the items appear in the HTML in the order 1-2-3, because item 2 has its order property set to 3, the actual rendered order is 1-3-2.

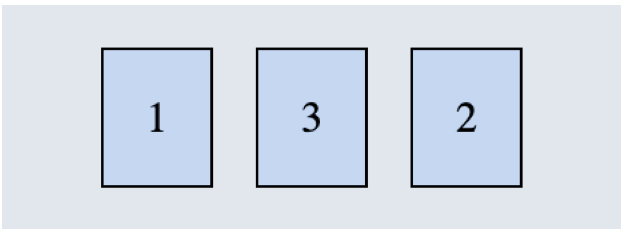


Figure 10-29. *The reordered items*

Multiple flex items can have the same value for the `order` property. If more than one item has the same value, those items will be laid out in the order they appear in the source HTML.

Accessibility Considerations

The `order` property only affects the displayed order on screen. It does not affect the order of the elements in other contexts. A screen reader, for example, will read the elements in their source order, not the displayed order.

This makes the display of the items “out of sync” from how they are displayed on screen, which could be confusing. For this reason, the `order` property is best used sparingly.

Flexbox Layout Use Cases

We’ve looked at the flexbox layout properties and behavior in detail. Now, let’s look at a few real-world use cases for flexbox layouts.

Absolute Centering

With flexbox, the problem of absolute (horizontal and vertical) centering is easily solved by setting both `align-items` and `justify-content` to `center`. Listing 10-12 shows how this can be used to center content within an element.

Listing 10-12. Absolute centering with flexbox

```
<style>
  .container {
    background: skyblue;
    display: flex;
    align-items: center;
    justify-content: center;
    width: 200px;
    height: 100px;
  }
</style>

<div class="container">
  <div>Centered text</div>
</div>
```

This centers the content both horizontally and vertically, as Figure 10-30 shows.

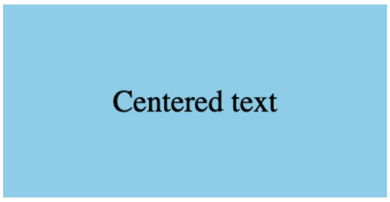


Figure 10-30. *Absolute centering of text within a flex container*

A Flexbox-Based Page Layout

Nesting flexbox layouts allows you to create all kinds of layouts. Listing 10-13 is an example of creating a full-page layout using nested flexbox layouts.

Listing 10-13. A full-page layout with flexbox

```
<style>
  body {
    margin: 0;
  }

  .container {
    display: flex;
    flex-direction: column;
    height: 100vh;
    box-sizing: border-box;
  }

  .header {
    background: #94a3b8;
    padding: 1rem;
  }

  .main {
    display: flex;
    flex-direction: row;
    flex-grow: 1;
  }
  .sidebar {
    background: #bfdbfe;
    padding: 1rem;
  }

  .content {
    flex-grow: 1;
    background: #ffffff;
    padding: 1rem;
  }

  .sidebar-2 {
    background: #bfdbfe;
    padding: 1rem;
  }

  .footer {
    background: #94a3b8;
    padding: 1rem;
  }
</style>

<div class="container">
  <header class="header">Header</header>
  <main class="main">
    <div class="sidebar">Sidebar</div>
    <div class="content">Content</div>
    <div class="sidebar-2">Sidebar 2</div>
  </main>
  <footer class="footer">Footer</footer>
</div>
```

This code results in the layout shown in Figure 10-31.

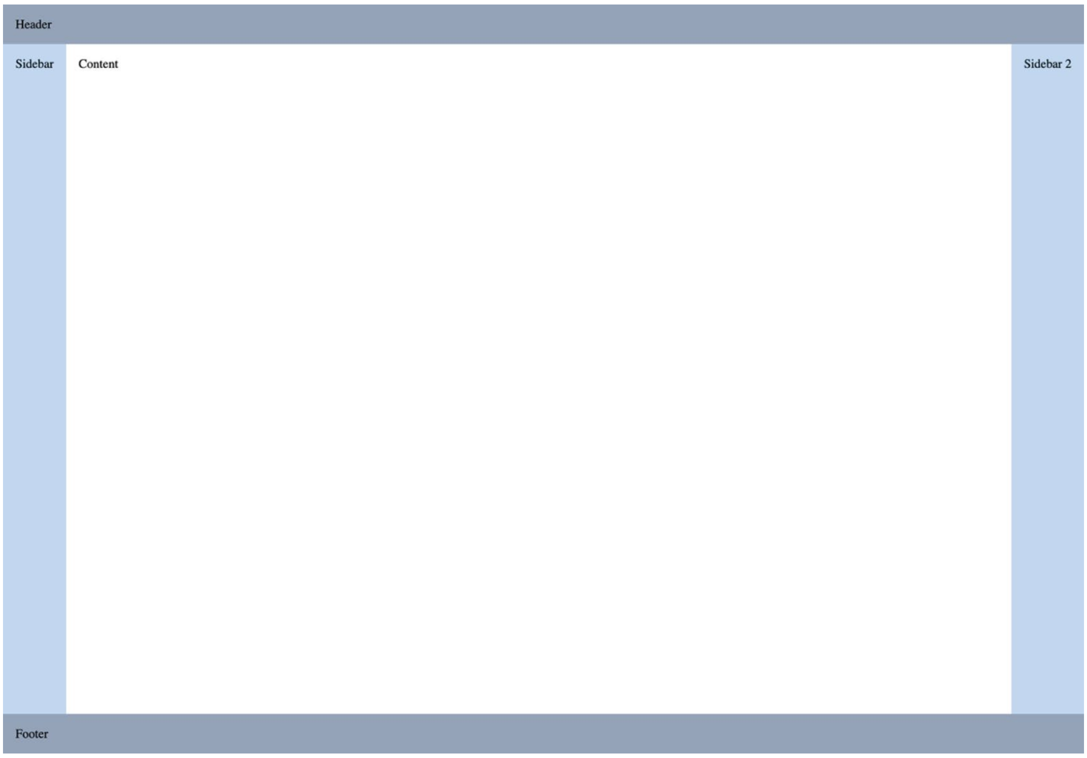


Figure 10-31. *The full-page flexbox layout*

The root container defines a `column` flexbox layout. It contains a `header`, a `main` element which contains the other content, and a `footer`. The `main` element gets a `flex-grow` of `1`. This means that while the `header` and `footer` use just enough space for their content, the `main` element will grow to fill the remaining space.

Then, the `main` element has a `row` flexbox layout. The two sidebars on either end just use their natural width, and the content area in the middle again uses a `flex-grow` of `1` to fill horizontally.

We could have even deeper nesting. For example, the `header` could contain a `row` flexbox layout with navigation links, or the sidebar could contain a `column` flexbox layout.

Summary

- A flexbox layout is made up of a flex container and its child flex items.
- A flexbox layout is one-dimensional; it's either a single row or a single column.
- An element becomes a flex container by setting its display property to `flex` or `inline-flex`.
- If flex items are too large to fit their container, the browser will try to shrink them to fit. This shrinking behavior can be changed with the `flex-shrink` property.
- Flex items can be set to grow to fill available space using the `flex-grow` property.
- The alignment and distribution of space are controlled by setting the `align-items` and `justify-content` properties.
- Multiple flexbox layouts can be nested.
- Flexbox can be used to absolutely center an element inside its container.

CHAPTER 11

CSS Grid

In Chapter 10, we looked at flexbox layouts in depth and have seen how powerful they can be. Still, there are limitations. Flexbox is a one-dimensional layout where items are arranged horizontally or vertically in rows or columns.

CSS Grid allows you to create two-dimensional grid-based layouts with rows and columns. It's supported in all modern browsers.

Basic Concepts

Let's start with the basic concepts of CSS Grid layouts.

Grid Container

The *grid container* is the outer element that contains the grid layout. All its direct children are grid items. To make an element a grid container, set its `display` property to `grid` or `inline-grid`. The difference is an element with `display: grid` will be a block element, while an element with `display: inline-grid` will be an inline element.

Grid Item

All immediate children of the grid container are *grid items*. Beyond the immediate children, descendant elements are not grid items. No special CSS properties need to be applied to make an item a grid item. The child elements automatically become grid items, and by default they are laid out in the order that they appear in the HTML markup.

J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_11

Grid Lines

Grid lines divide the rows and columns of the grid. The grid lines are numbered starting with 1, as Figure 11-1 shows.

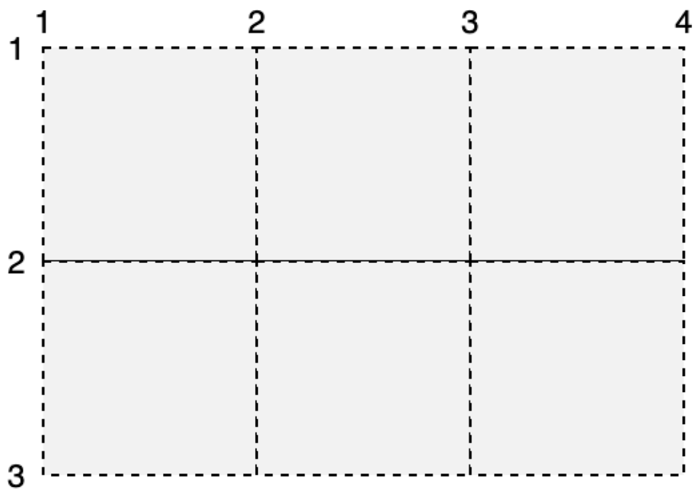


Figure 11-1. *Grid lines*

Grid Tracks

The *tracks* of the grid are the rows and columns between the grid lines. The tracks are what contain the grid items. Like with grid lines, the numbering of tracks starts at 1. In Figure 11-2, row track 1 is highlighted.

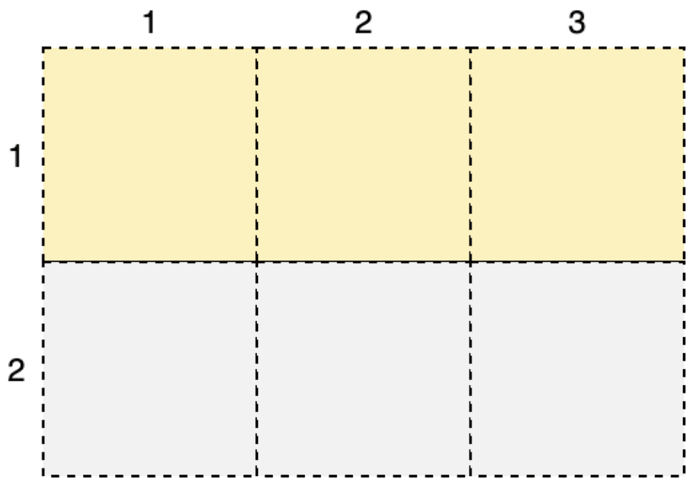


Figure 11-2. *Grid tracks*

Grid Areas

A grid area is the space enclosed by any four grid lines. It can contain a single cell or multiple cells. Figure 11-3 shows a highlighted grid area that takes up four cells.

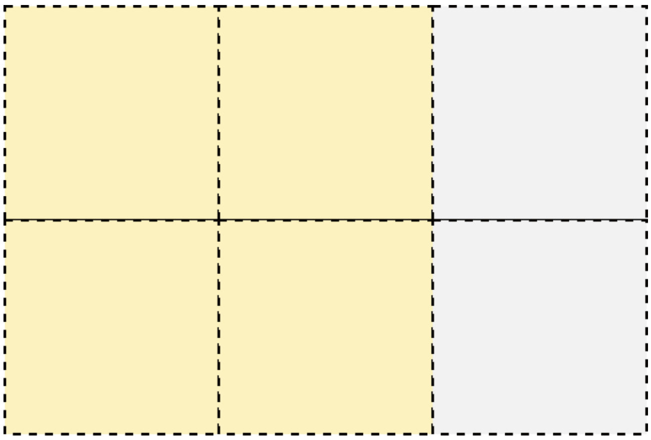


Figure 11-3. *A grid area*

Explicit and Implicit Grids

When grid rows and columns are explicitly defined with CSS properties such as `grid-template-rows` and `grid-template-columns`, this is known as the *explicit grid*.

If more items are added than are accounted for in the explicit grid, the grid layout creates additional rows and/or columns to fit these extra items. This is the *implicit grid*.

The `fr` Unit

Grid sizes can be specified with any of the units we've looked at so far - `px`, `em`, `rem`, even percentages. CSS Grid introduces a new unit, the `fr` unit. This unit refers to a fraction of the free space within the grid. For example, if there are four columns each at a width of 1fr, then each column will take up 25% of the total width of the grid.

If a grid has two columns, one `1fr` and one `2fr`, the second column will take up twice as much of the free space as the first column.

Defining a Grid Layout

To define the rows and columns of an explicit grid, use the `grid-template-rows` and `grid-template-columns` properties. These are used to specify the heights of the rows and the widths of the columns, respectively. In other words, they are used to specify the size of the grid tracks.

Like flexbox, CSS Grid layouts support the `gap` property which defines the spacing between grid items. You can use the `gap` property to set the same spacing between rows and columns, or you can use the individual `row-gap` and `column-gap` properties to use different gap sizes between rows and columns.

Listing 11-1 defines a basic grid layout.

Listing 11-1. A basic grid

```
<style>
  .container {
    display: grid;
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr 1fr;
    gap: 8px;
    width: 300px;
    border: 1px solid #000000;
    padding: 8px;
  }

  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
</div>
```

Figure 11-4 shows the resulting grid layout.

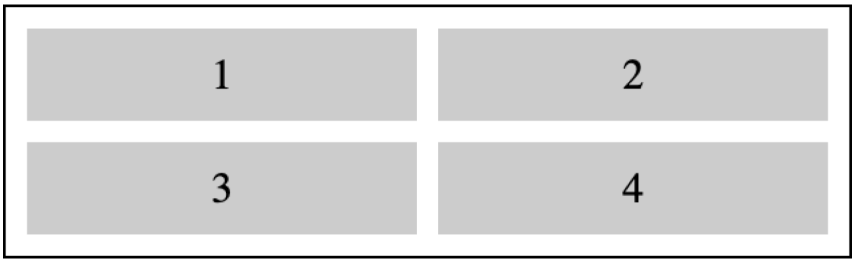


Figure 11-4. *The rendered grid*

We have defined an explicit grid with two `1fr` column tracks and two `1fr` row tracks. As you can see, you don't need to specify a row or column for the grid items – by default, the grid container places its items in order, starting at the first column of the first row.

Because each grid row and column is defined as `1fr`, the rows and columns are all equally sized.

Right now, we have two rows and two columns, a total of four grid items. What happens if we add more grid items to the container? How does this affect the grid layout? Let's do this in Listing 11-2.

Listing 11-2. Adding two more grid items

```
<style>
  .container {
    display: grid;
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr 1fr;
    gap: 8px;
    width: 300px;
    border: 1px solid #000000;
    padding: 8px;
  }
  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
</div>
```

The updated grid is shown in Figure 11-5.

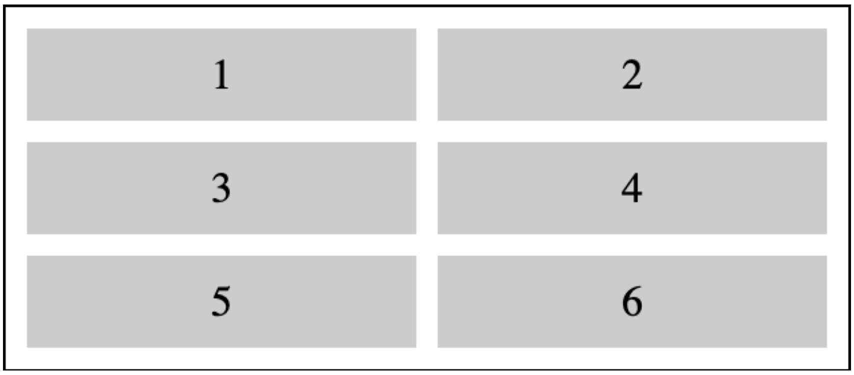


Figure 11-5. *The updated grid*

Items 5 and 6 are added to the grid. The grid now has three rows, even though `grid-template-rows` only defines two rows. Items 5 and 6 make up the implicit grid. The implicit grid layout continues the layout that was set by the explicit grid. Since two columns were defined, items 5 and 6 are in a new row across these existing columns.

Grid Sizing

Each row in the previous grid layout is the same height, since they are all set to `1fr`. If we set the second row to `2fr`, it will take up twice as much space as the others. Listing 11-3 demonstrates this.

Listing 11-3. Adjusting the grid sizing

```
<style>
  .container {
    display: grid;
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr 2fr 1fr;
    gap: 8px;
    width: 300px;
    border: 1px solid #000000;
    padding: 8px;
  }

  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
</div>
```

Figure 11-6 shows the updated layout.

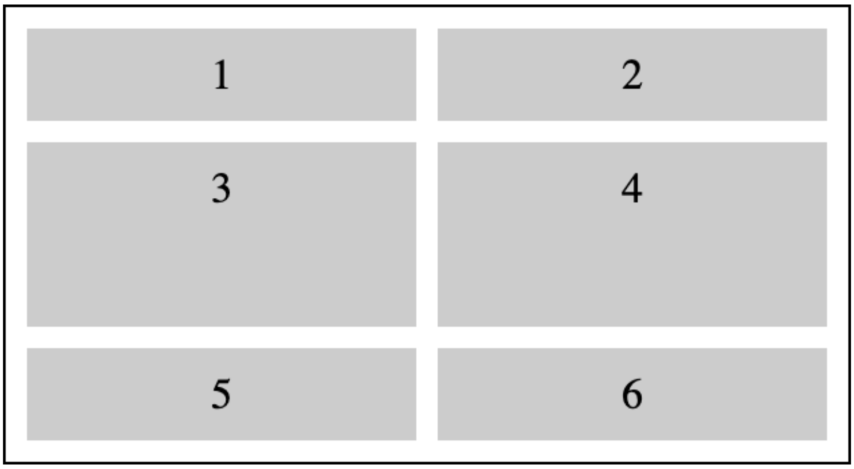


Figure 11-6. *The updated grid with larger second row*

You can mix the `fr` unit with other units, too. Listing 11-4 shows a grid that uses mixed units.

Listing 11-4. Mixing units

```
<style>
  .container {
    display: grid;
    grid-template-columns: 100px 1fr 3rem;
    grid-template-rows: 1fr 1fr;
    gap: 8px;
    width: 400px;
    border: 1px solid #000000;
    padding: 8px;
  }

  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
</style>
```

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
</div>
```

Figure 11-7 shows the resulting layout.

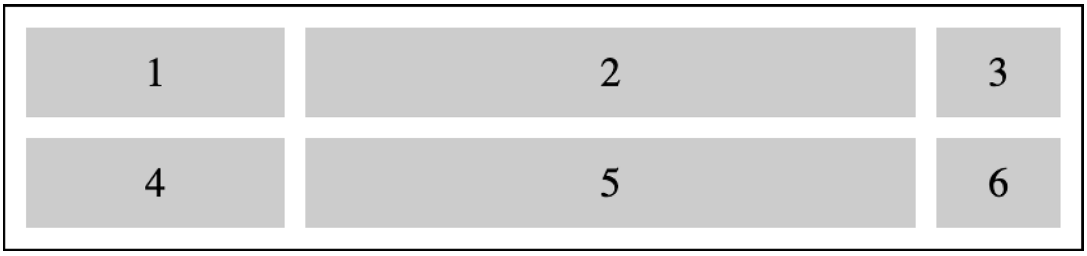


Figure 11-7. *A grid using different units for its columns*

Grid Sizing Functions

CSS is full of useful functions, and CSS Grid is no exception. Here are a few functions that can help you when defining your grid rows or columns.

The `repeat` Function

Sometimes, defining grid sizes can be repetitive, as seen in Listing 11-5.

Listing 11-5. A repetitive grid definition

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr;
  grid-template-rows: 5rem 5rem;
}
```

In the `grid-template-columns` and `grid-template-rows` properties, we are repeating values multiple times. For cases like this, we can use the `repeat` function instead, as Listing 11-6 shows.

Listing 11-6. Using the `repeat` function

```
.container {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  grid-template-rows: repeat(2, 5rem);
}
```

The `minmax` Function

If you use a fixed size for a row or column, and the cell's content does not fit within the cell, the content will overflow. This is demonstrated in Listing 11-7.

Listing 11-7. A grid with overflowing cell content

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    grid-template-rows: repeat(2, 2rem);
    gap: 8px;
    padding: 8px;
    width: 400px;
    background: #eeeeee;
  }

  .item {
    background: #cccccc;
    text-align: center;
  }
</style>
```

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">A really long value to see what happens</div>
  <div class="item">6</div>
  <div class="item">7</div>
  <div class="item">8</div>
</div>
```

Figure 11-8 shows the result. The content in cell 5 doesn't fit within a `2rem` height, so it overflows.

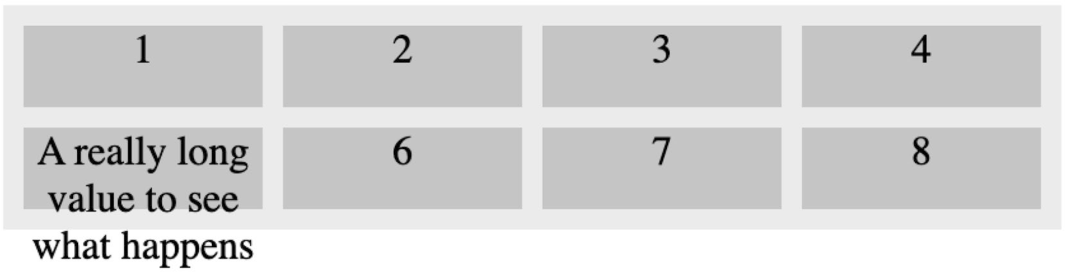


Figure 11-8. *Overflowing cell content*

To fix this, you can use the `minmax` function. The function takes two arguments, the minimum size and the maximum size. Listing 11-8 has an example of this.

Listing 11-8. The `minmax` function

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(4, 1fr);
    grid-template-rows: repeat(2, minmax(2rem, min-content));
    gap: 8px;
    padding: 8px;
    width: 400px;
    background: #eeeeee;
  }

  .item {
    background: #cccccc;
    text-align: center;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">A really long value to see what happens</div>
  <div class="item">6</div>
  <div class="item">7</div>
  <div class="item">8</div>
</div>
```

Figure 11-9 shows the new grid.

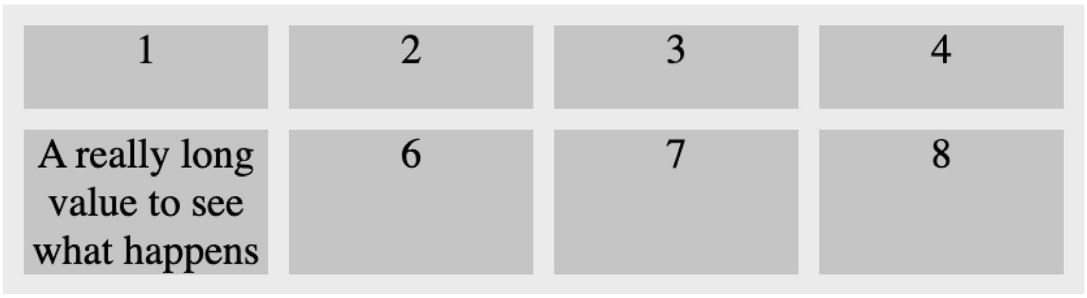


***Figure 11-9.** The overflow is fixed*

The rows now have a minimum height of `2rem`, but because we also specify a maximum of `min-content`, the row can grow to fit the content if necessary. This fixes the overflow issue from Figure 11-8.

Sizing with `auto-fill`

Sometimes, you might not want to specify an exact number of rows or columns in a grid. You might want to fit as many columns as will fit into the container's width. For this, you can use the `repeat` function with the `auto-fill` keyword, as Listing 11-9 shows.

Listing 11-9. Using the auto-fill keyword

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(auto-fill, 5rem);
    gap: 8px;
    padding: 8px;
    background: #eeeeee;
  }

  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
  <div class="item">7</div>
</div>
```

If the viewport is wide enough, all seven grid items will appear in a single row, as Figure 11-10 shows. The columns are `5rem` in size. New columns are added for each item that fits in the row.

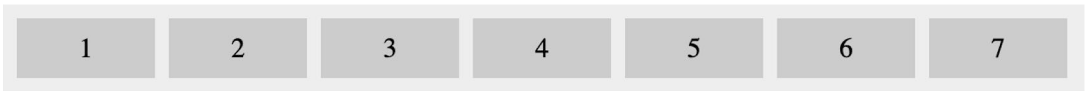


Figure 11-10. *The grid in a wide viewport*

However, if you resize the window so that it's no longer wide enough to fit all seven grid items, the grid uses as many columns as will fit in the available space. The remaining grid items wrap to subsequent rows. Figure 11-11 shows the multi-row grid.

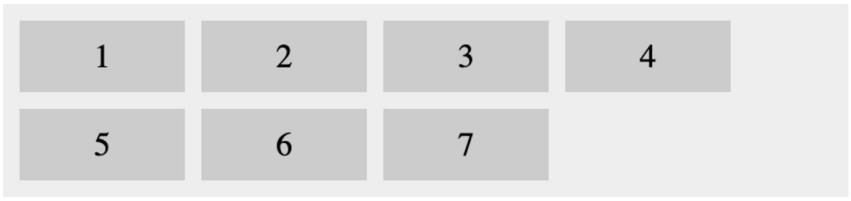


Figure 11-11. *The grid items automatically wrap with `auto-fill`*

`auto-fill` can also be combined with the `minmax` function. Note the gap in Figure 11-11 after the last column. This is part of the grid container, but there is no column there. In Listing 11-10, we add the `minmax` function to the column sizing.

Listing 11-10. Adding the `minmax` function

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(5rem, 1fr));
    gap: 8px;
    padding: 8px;
    background: #eeeeee;
  }

  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
</style>
```

```
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
  <div class="item">7</div>
</div>
```

Adding the minmax function, with a `1fr` maximum, allows the columns to grow equally to use the extra space. This way, there is no gap of wasted space, as Figure 11-12 shows.

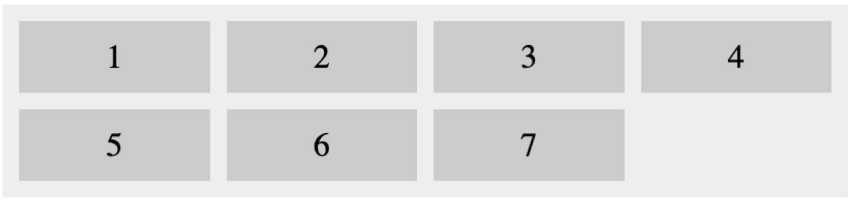


Figure 11-12. *The grid using `auto-fill` and `minmax`*

Sizing with `auto-fit`

The `auto-fit` keyword behaves differently than `auto-fill`. With `auto-fit`, grid items are sized to fit within the available space without adding more columns. Let's compare two grid layouts. First, Listing 11-11 has another example of using auto-fill when there is extra horizontal space available.

Listing 11-11. A grid with `auto-fill`

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(5rem, 1fr));
    gap: 8px;
    padding: 8px;
    background: #eeeeee;
  }
  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
  <div class="item">7</div>
</div>
```

As Figure 11-13 shows, there is extra space after the seven grid cells. The browser creates extra empty cells at the end of the grid row, which accounts for the extra space.

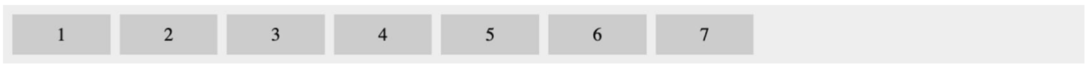


Figure 11-13. *The grid with the `auto-fill` keyword*

Listing 11-12 shows the same layout but uses `auto-fit` instead of `auto-fill`.

Listing 11-12. Using the `auto-fit` keyword

```
<style>
    .container {
      display: grid;
      grid-template-columns: repeat(auto-fit, minmax(5rem, 1fr));
      gap: 8px;
      padding: 8px;
      background: #eeeeee;
    }
    .item {
      background: #cccccc;
      text-align: center;
      padding: 8px;
    }
  </style>

  <div class="container">
    <div class="item">1</div>
    <div class="item">2</div>
    <div class="item">3</div>
    <div class="item">4</div>
    <div class="item">5</div>
    <div class="item">6</div>
    <div class="item">7</div>
  </div>
```

Figure 11-14 shows the resulting grid layout. With auto-fit, the seven columns grow to fill the available space.

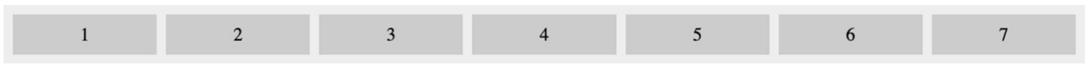


***Figure 11-14.** The updated grid layout using `auto-fit`*

Grid Positioning

By default, the grid items are placed automatically, filling each column, then each row. There are options that can be used to customize the positioning of grid items.

Specifying the Row and Column

You can override the default grid positioning and specify specific grid row and column indices for grid items using the `grid-row` and `grid-column` properties. Row and column numbers start at 1. Listing 11-13 shows an example of a grid with one of the grid items positioned manually using `grid-row` and `grid-column`.

Listing 11-13. Specifying the row and column for one item

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    background: #eeeeee;
    padding: 8px;
    gap: 8px;
    width: 300px;
  }

  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }

  .item1 {
    grid-row: 2;
    grid-column: 2;
    background: skyblue;
  }
</style>

<div class="container">
  <div class="item item1">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
</div>
```

Figure 11-15 shows the resulting grid.

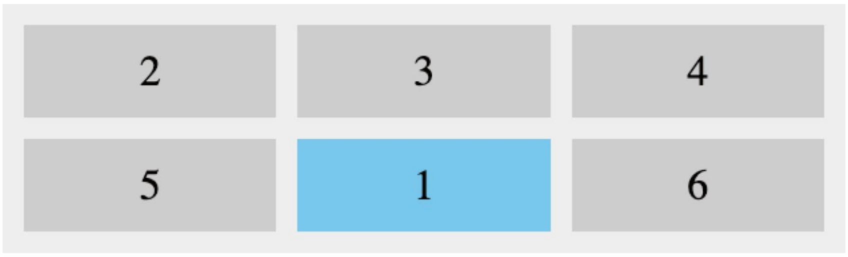


Figure 11-15. *The rendered grid*

Grid item 1 has its row and column set to 2, and it's positioned accordingly in the grid. The rest of the grid items are placed automatically around this item.

Spanning Multiple Rows or Columns

Grid items can also span across multiple rows or columns. This is controlled by the `grid-row-start`, `grid-row-end`, `grid-column-start`, and `grid-column-end` properties. These properties reference the grid line numbers, not cell numbers. In Listing 11-14, the blue grid item spans across two rows and two columns.

Listing 11-14. Spanning multiple grid cells

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    background: #eeeeee;
    padding: 8px;
    gap: 8px;
    width: 300px;
  }

  .item {
    background: #cccccc;
    text-align: center;
    padding: 8px;
  }
  .item1 {
    grid-row-start: 1;
    grid-row-end: 3;
    grid-column-start: 1;
    grid-column-end: 3;
    background: skyblue;
  }
</style>

<div class="container">
  <div class="item item1">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
  <div class="item">5</div>
  <div class="item">6</div>
</div>
```

Figure 11-16 shows the resulting grid behavior.

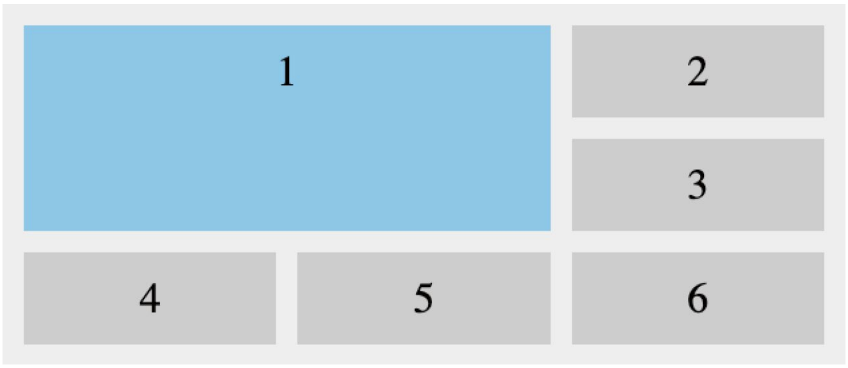


Figure 11-16. *Grid item 1 spans two rows and two columns*

Grid item 1 starts at column line 1 and ends at column line 3, making it span two columns. Similarly, it starts at row line 1 and ends at row line 3, making it span two rows.

The `grid-row` or `grid-column` properties can also be used as a shorthand for these properties. The expected format is the starting grid line, a slash, and the ending line. Listing 11-15 applies the same behavior as Listing 11-14, using the shorthand syntax.

Listing 11-15. Using the shorthand properties

```
.item1 {
  grid-row: 1 / 3;
  grid-column: 1 / 3;
  background: skyblue;
}
```

Instead of specifying an ending row or column line number, we can also use the `span` keyword. This specifies that the item spans that number of rows or columns, starting at the specified start index. Listing 11-16 shows the equivalent syntax using the `span` keyword.

Listing 11-16. Using the `span` keyword

```
.item1 {
  grid-row: 1 / span 2;
  grid-column: 1 / span 2;
  background: skyblue;
}
```

Named Grid Lines

Grid lines can be referenced by their numerical index, as we have already seen. But we can also assign names to the grid lines and reference those names instead. The grid lines are named within a `grid-template-rows` or `grid-template-columns` expression, in between the grid track definitions. The grid line names are placed inside square brackets.

These grid line names can then be referenced from the `grid-row` and `grid-column` properties. Listing 11-17 defines a basic page layout defined using named grid lines.

Listing 11-17. Using named grid lines

```
<style>
  .container {
    display: grid;
    gap: 5px;
    width: 500px;
    grid-template-rows:
      [header-start] 2rem
      [content-start] 10rem
      [footer-start] 2rem
      [footer-end];
    grid-template-columns:
      [sidebar-start] 5rem
      [content-start] 1fr
      [content-end];
  }

  .container > div {
    background: lightgray;
  }

  .header {
    grid-row: header-start / content-start;
    grid-column: sidebar-start / content-end;
  }

  .footer {
    grid-column: sidebar-start / content-end;
  }
</style>

<div class="container">
  <div class="item header">Header</div>
  <div class="item sidebar">Sidebar</div>
  <div class="item content">Content</div>
  <div class="item footer">Footer</div>
</div>
```

Figure 11-17 shows the resulting layout.

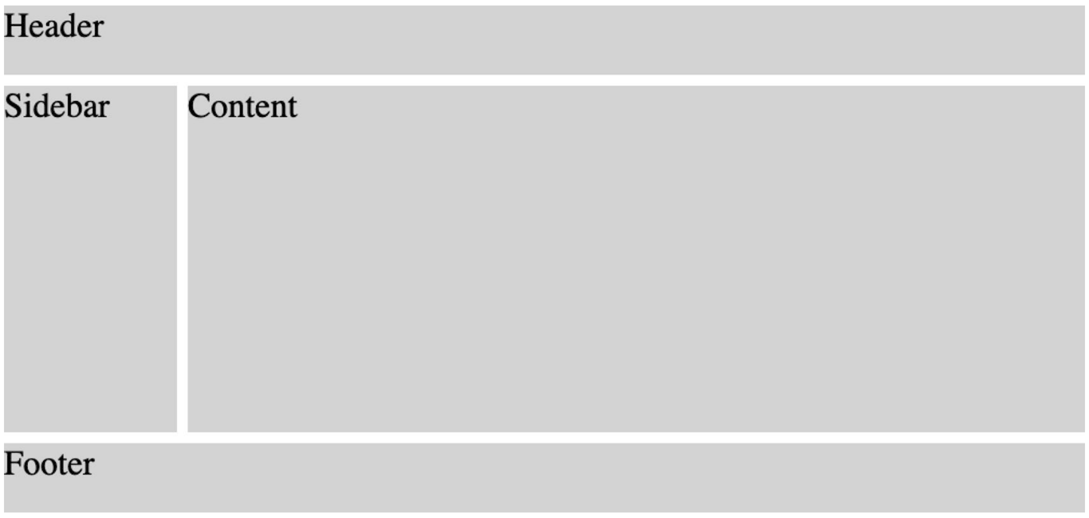


Figure 11-17. *Basic layout defined with named grid lines*

Named Grid Areas

In addition to named grid lines, you can also define named grid areas. This allows us to place grid items in the desired areas without having to specify start and end lines. The areas are defined with the `grid-template-areas` property. If two adjacent areas have the same name, then an item placed in that area will span those areas.

Note that `grid-template-areas` does not define the size of the grid tracks, it just defines the arrangement of the regions. To set row and column sizing, you can use `grid-template-rows` and `grid-template-columns` as before. To place a grid item in a particular named grid area, you can use the `grid-area` property and specify the desired area name.

Listing 11-18 defines the same layout as the one from Listing 11-17 but using named grid areas instead of named grid lines.

Listing 11-18. Building a layout with named grid areas

```
<style>
  .container {
    display: grid;
    grid-template-rows: 2rem 10rem 2rem;
    grid-template-columns: 5rem 1fr;
    grid-template-areas:
      "header header"
      "sidebar content"
      "footer footer";
    gap: 5px;
    width: 500px;
  }

  .header {
    grid-area: header;
  }

  .footer {
    grid-area: footer;
  }

  .sidebar {
    grid-area: sidebar;
  }

  .content {
    grid-area: content;
  }

  .container > div {
    background: lightgray;
  }
</style>

<div class="container">
  <div class="item header">Header</div>
  <div class="item sidebar">Sidebar</div>
  <div class="item content">Content</div>
  <div class="item footer">Footer</div>
</div>
```

As Figure 11-18 shows, the resulting layout is the same.

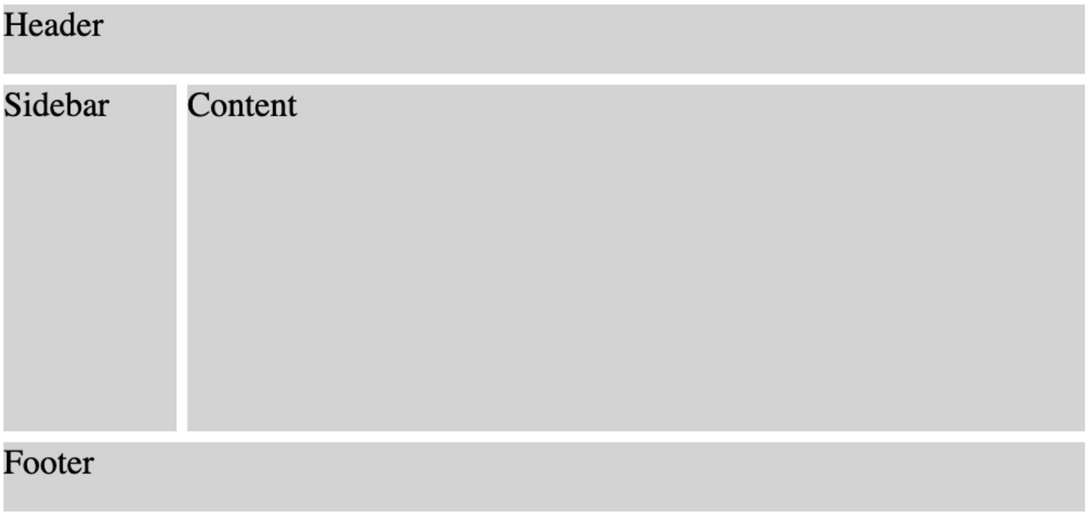


***Figure 11-18.** The layout with named grid areas*

Grid Alignment

Just like with flexbox, CSS Grid gives you total control over the alignment of items when they don't fill their container. There are several properties that control this alignment.

`justify-items`

The `justify-items` property defines how grid items are aligned along the row axis when there is extra space left inside a grid cell. This property is defined on the container and applies to all the items within that container. Supported values include

- `stretch`: Stretches the item content along the row axis to fill the grid cells. This is the default and is shown in Figure 11-19.

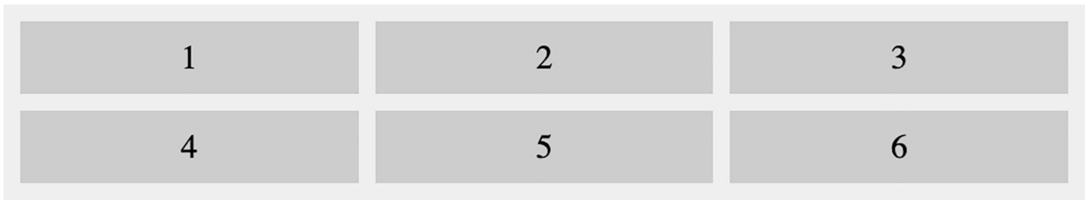


***Figure 11-19.** The items stretched to fill the grid cells*

- `start`: Items are aligned with the starting edge of the cells along the row axis. This is shown in Figure 11-20.

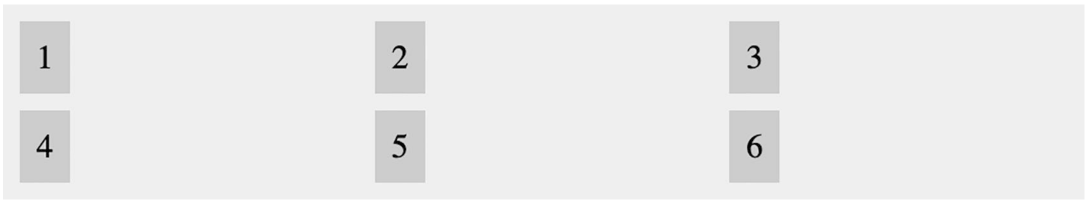


Figure 11-20. *The items aligned at the start of the cells*

- `end`: Items are aligned with the ending edge of the cells along the row axis. This is shown in Figure 11-21.

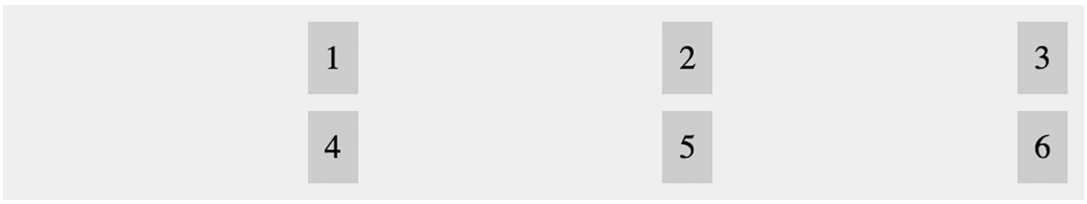


Figure 11-21. *The items aligned at the end of the cells*

- `center`: Items are centered along the row axis in their cells. This is shown in Figure 11-22.

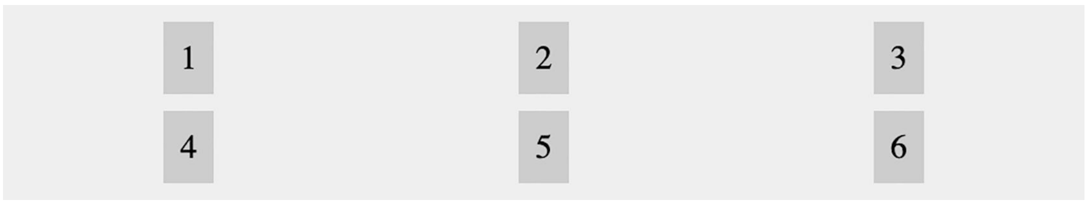


Figure 11-22. *The items aligned at the center of the cells*

`align-items`

The `align-items` property defines how grid items are aligned in the opposite direction, along the column axis. Like `justify-items`, this property is set on the container. The supported values are

- `stretch`: Items are stretched along the column axis to fill the entire height of the cells. This is the default and is shown in Figure 11-23.

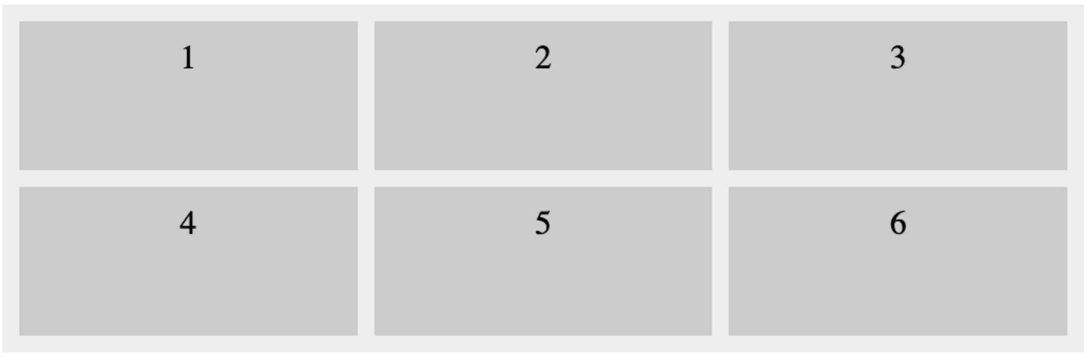


***Figure 11-23.** The items stretched to fill the grid cells*

- `start`: Items are aligned at the top edge of the cells. This is shown in Figure 11-24.

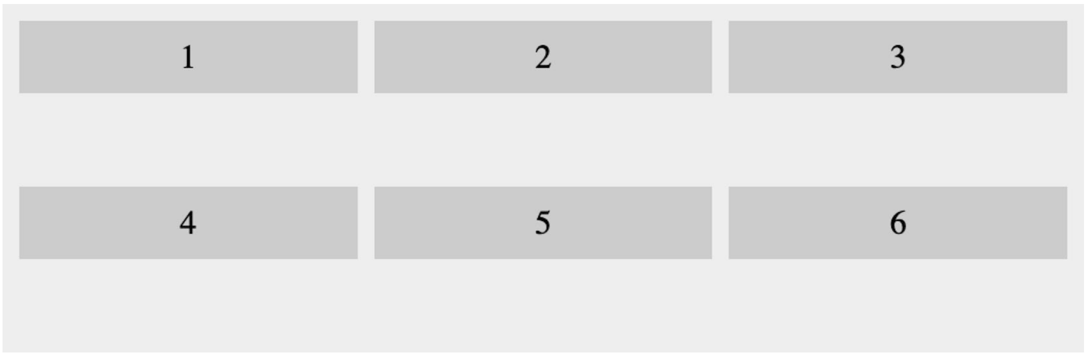


***Figure 11-24.** The items aligned at the top edge of the grid cells*

- `end`: Items are aligned at the bottom edge of the cells. This is shown in Figure 11-25.

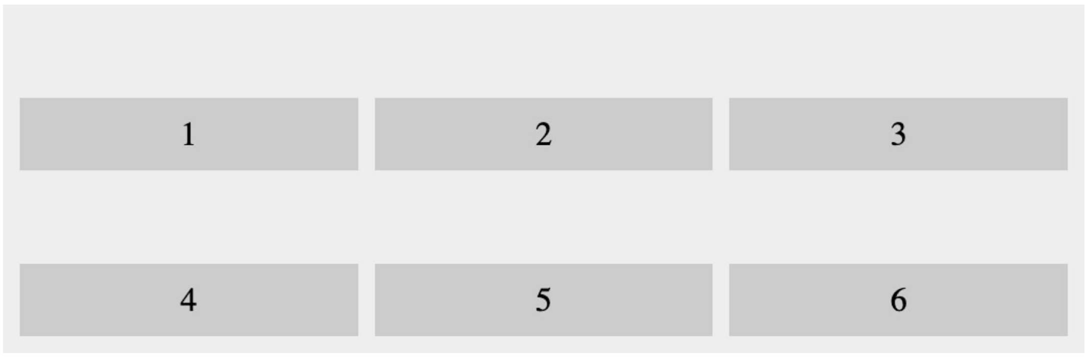


Figure 11-25. *The items aligned at the bottom edge of the grid cells*

- `center`: Items are aligned at the center of the cells. This is shown in Figure 11-26.

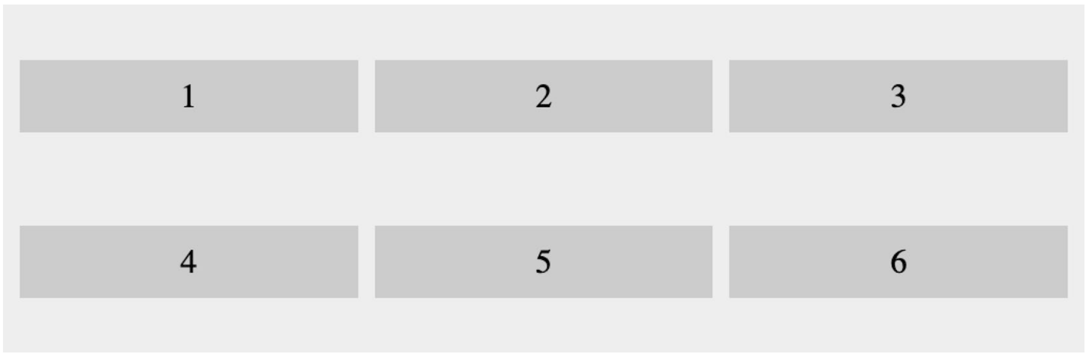


Figure 11-26. *The items aligned at the center of the grid cells*

`justify-content`

If the grid rows and/or columns aren't sized with relative/flexible units like `fr`, we could end up with empty space in the grid container. If this happens, the `justify-content` property defines where within the grid container the grid items will be aligned along the row axis. It supports several values:

- `start`: Aligns the grid at the start of the row axis. This is shown in Figure 11-27.

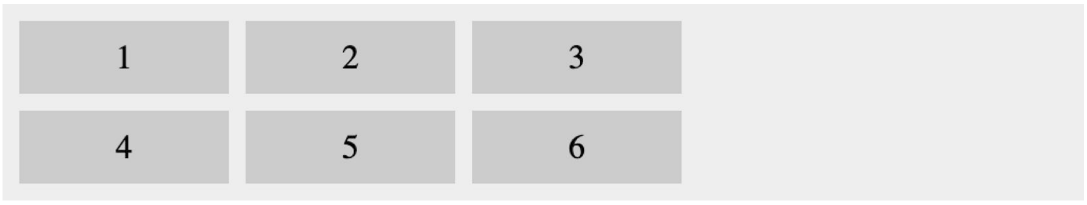


Figure 11-27. *The grid aligned at the start of the row axis*

- `end`: Aligns the grid at the end of the row axis. This is shown in Figure 11-28.

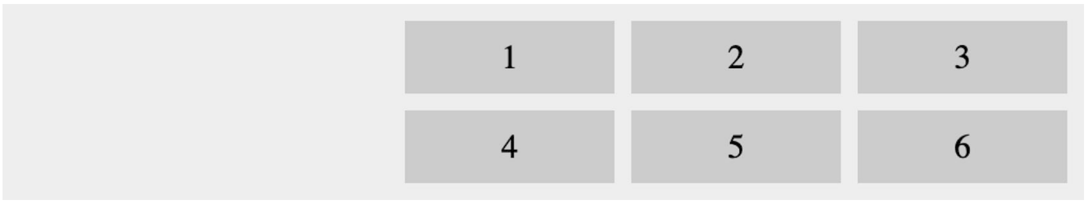


Figure 11-28. *The grid aligned at the end of the row axis*

- `center`: Aligns the grid at the center of the row axis. This is shown in Figure 11-29.

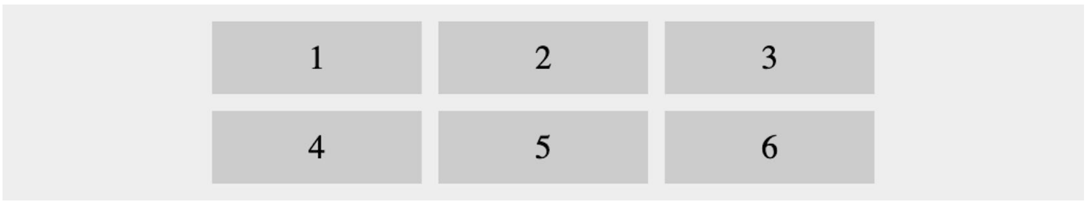


Figure 11-29. *The grid aligned at the center of the row axis*

- `space-around`: Adds even spacing between columns, with half-sized spaces at the start and end. This is shown in Figure 11-30.

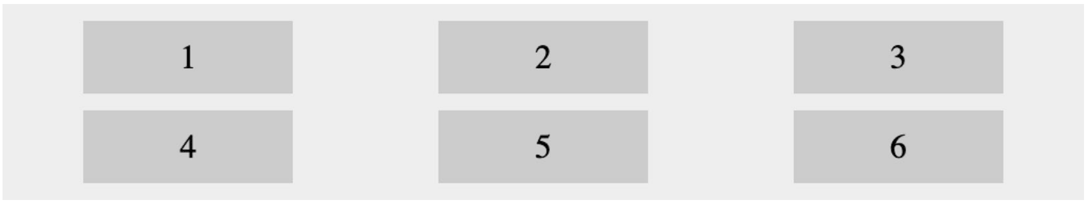


Figure 11-30. *Spacing between the columns*

- `space-evenly`: Adds even spacing between columns, with full-sized spaces at the start and end. This is shown in Figure 11-31.

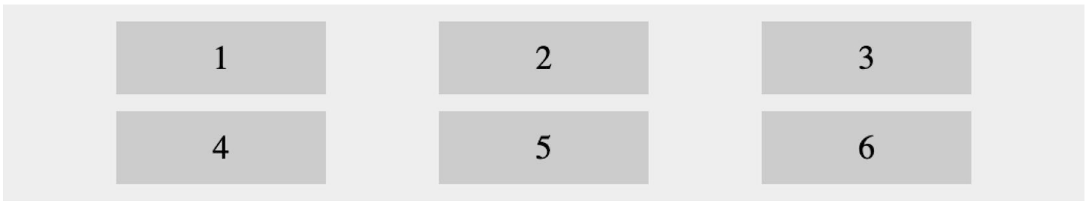


Figure 11-31. *Even spacing between and around the columns*

- `space-between`: Places the first column flush with the start of the container, the last column flush with the end of the container, and adds even spacing between the other columns. This is shown in Figure 11-32.

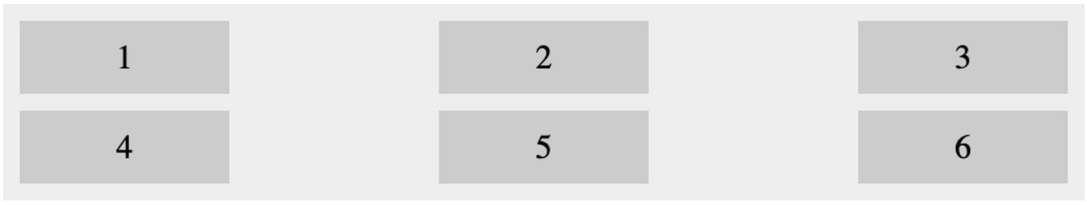


Figure 11-32. *Maximized spacing between the columns*

`align-content`

`align-content` is like `justify-content`, only it determines how the grid rows are aligned along the column axis instead of columns along the row axis. The supported values include

- `start`: Aligns the grid rows at the start of the column axis. This is shown in Figure 11-33.

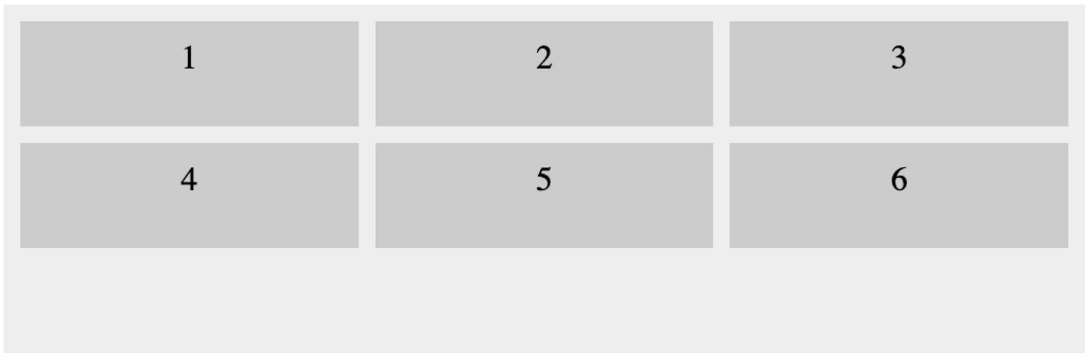


***Figure 11-33.** Rows aligned at the start of the column axis*

- `end`: Aligns the grid rows at the end of the column axis. This is shown in Figure 11-34.

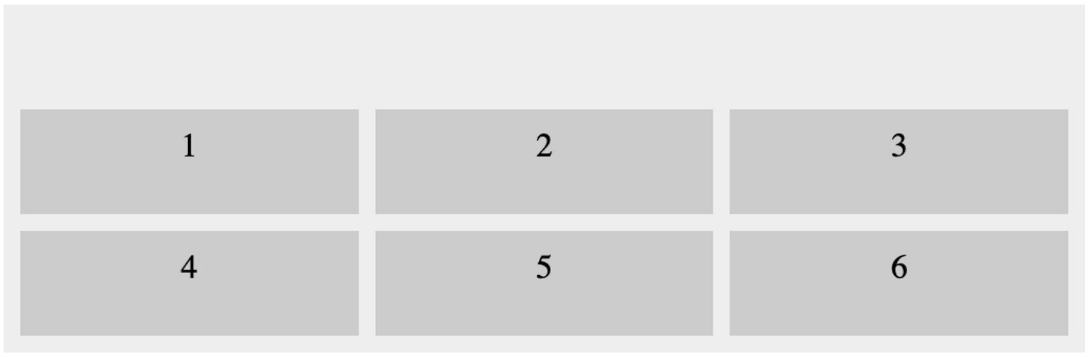


***Figure 11-34.** Rows aligned at the end of the column axis*

- `center`: Aligns the grid rows at the center of the column axis. This is shown in Figure 11-35.

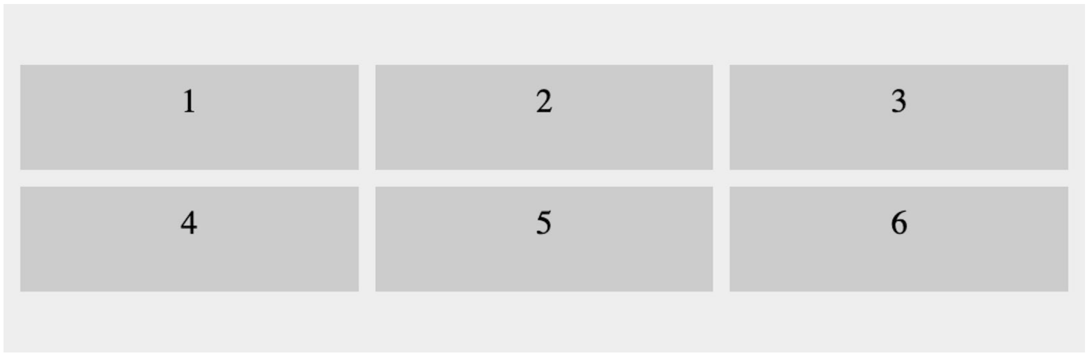


***Figure 11-35.** Rows aligned at the center of the column axis*

- `space-around`: Adds even spacing between rows, with half spacing at the start and end. This is shown in Figure 11-36.

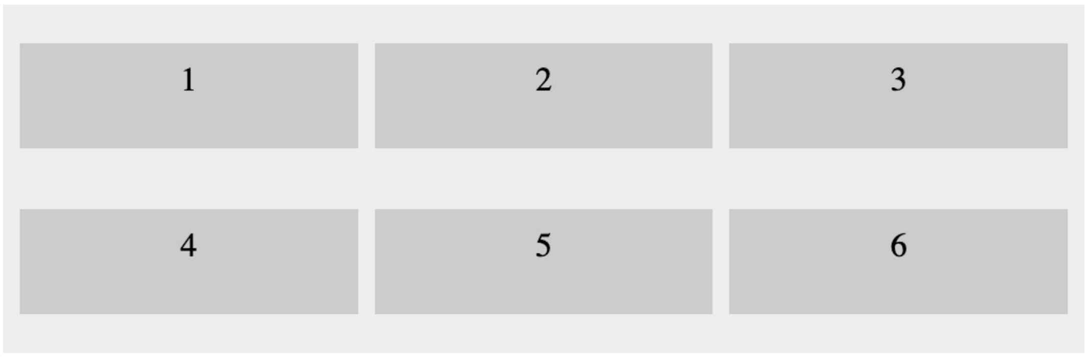


***Figure 11-36.** Spacing between and around the rows*

- `space-evenly`: Adds even spacing between rows, with full-sized spaces at the start and end. This is shown in Figure 11-37.

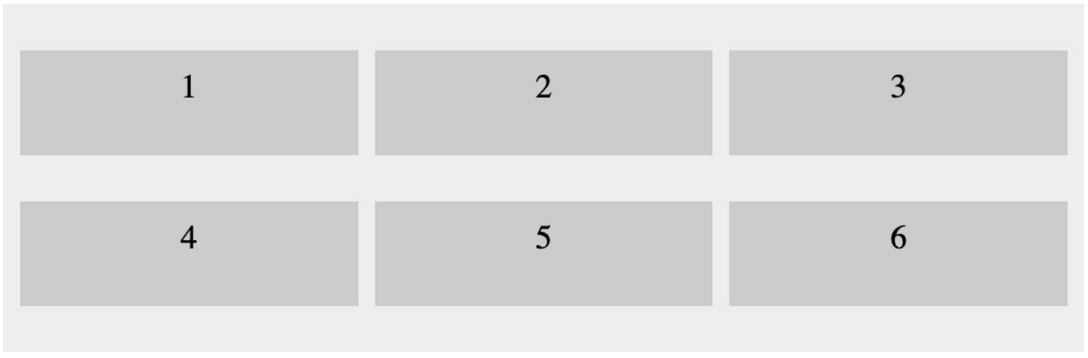


***Figure 11-37.** Even spacing between and around the rows*

- `space-between`: Places the first row flush with the top of the container, the last row flush with the bottom of the container, and adds even spacing in between the other rows. This is shown in Figure 11-38.

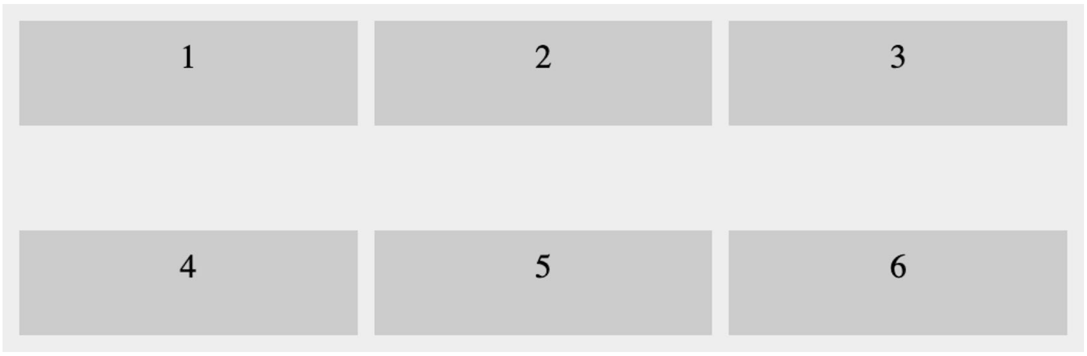


Figure 11-38. *Spacing between the rows*

Overriding for Individual Grid Items

The `justify-items` and `align-items` properties of the grid container define the alignment of the grid items along the row and column axis, respectively. These alignments can be overridden for an individual grid item by setting the `justify-self` and `align-self` properties, respectively.

CSS Subgrid

You can nest CSS Grid layouts inside other CSS Grid layouts. Sometimes, for more complex nested grid layouts, you might find that items from the child layouts don't line up nicely. Listing 11-19 shows a grid layout that contains two cards. Each card also uses a grid layout.

Listing 11-19. A grid-based card layout

```
<style>
  .grid {
    display: grid;
    grid-template-columns: repeat(2, 1fr);
    gap: 1rem;
  }
  .card {
    display: grid;
    grid-template-rows: auto 1fr;
    padding: 1rem;
    background: #eee;
  }
</style>

<div class="grid">
  <div class="card">
    <h2>Short Title</h2>
    <p>Short description.</p>
  </div>
  <div class="card">
    <h2>Longer Title That Wraps</h2>
    <p>
      Much longer description that takes more lines to explain everything
      properly.
    </p>
  </div>
</div>
```

Figure 11-39 shows the resulting card layout.

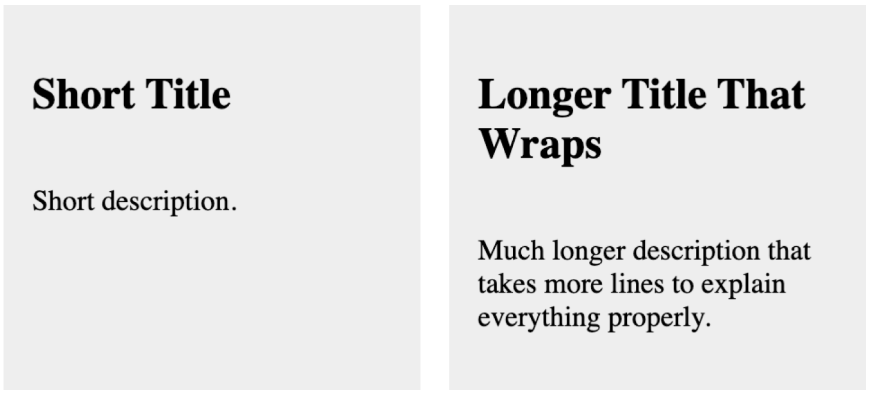

Short Title
Short description.
Longer Title That Wraps
Much longer description that takes more lines to explain everything properly.

Figure 11-39. *A grid-based card layout that uses nested grids*

Notice that the descriptions don't line up. The second card, with the wrapped title, bumps the description down further, and it's not aligned with the first card's description. We can solve this by using subgrids. Listing 11-20 implements this change.

Listing 11-20. Fixing the layout with subgrids

```
<style>
  .grid {
    display: grid;
    grid-template-columns: repeat(2, 1fr);
    grid-template-rows: auto auto;
    gap: 1rem;
  }

  .card {
    display: grid;
    grid-template-rows: subgrid;
    grid-row: span 2;
    background: #eee;
    padding: 1rem;
  }
</style>

<div class="grid">
  <div class="card">
    <h2>Short Title</h2>
    <p>Short description.</p>
  </div>
  <div class="card">
    <h2>Longer Title That Wraps</h2>
    <p>
      Much longer description that takes more lines to explain everything
      properly.
    </p>
  </div>
</div>
```

As Figure 11-40 shows, the descriptions now line up properly.

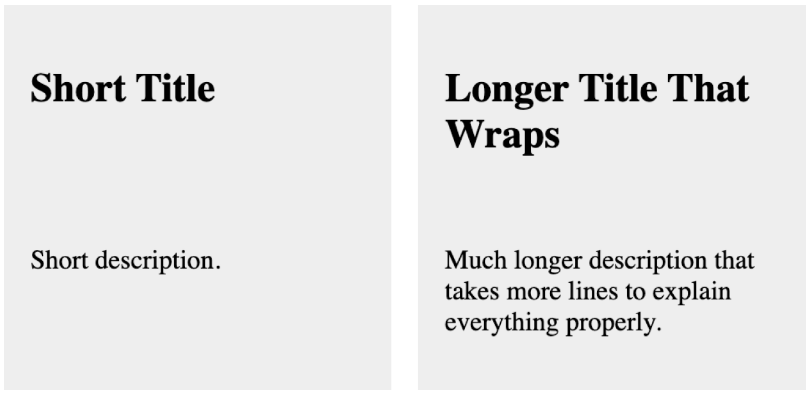

Short Title
Short description.
Longer Title That Wraps
Much longer description that takes more lines to explain everything properly.

Figure 11-40. *The card layout using subgrids*

These layouts are similar but have an important difference. In the first layout, from Listing 11-19, the parent grid has a single row. Then, the child grid in each card defines two rows: one for the title and one for the description. This causes a layout problem because the two nested card grids are totally independent of each other and have no way of lining up under a common layout.

In the second layout, from Listing 11-20, the two-row layout has been moved to the parent grid. Each card has a `grid-row: span 2` so that it takes up the full two rows of the parent grid.

Finally, the child grid layouts specify the `grid-template-rows` property using the special value `subgrid`. This makes the child layouts inherit the track sizes from the parent layout. Because they have common track sizes, the contents align perfectly.

Subgrid can be used with rows or columns. The key idea is that the child grid layouts inherit the row or column sizing from the parent layouts, and this makes it possible to ensure consistent alignment across components.

Summary

- An element can be made a grid container by setting its display property to `grid` or `inline-grid`.
- A grid container's immediate children become grid items.
- A grid has lines, tracks, and areas.

- The explicit grid is made up of the rows and columns defined by the `grid-template-rows` and `grid-template-columns` properties.
- The implicit grid contains any elements placed after all explicit grid areas are filled.
- The `fr` unit uses a fraction of the available free space.
- A grid item can span multiple rows and/or columns.
- Grid lines and areas can have names. These names can then be referenced when placing grid items.
- The `justify-items`, `align-items`, `justify-content`, and `align-content` properties define how extra space is handled in grids.
- An individual grid item can override its alignment with the `justify-self` and `align-self` properties.
- Subgrids are used with nested grid layouts. They make the child grid inherit the track sizing of its parent grid.

CHAPTER 12

Responsive Design

Responsive design is a technique for designing page layouts so that they are usable on devices with a variety of different screen sizes. Media queries and container queries are some of the main tools used for responsive design, as are flexbox and CSS Grid.

A layout that looks good on a large desktop display might not look so good on an iPhone. Media queries allow you to apply different CSS rules, or even entire stylesheets, depending on the size of the viewport. Additionally, container queries let you apply different styles based on the size of an element's container.

Media queries have other uses, too. For example, you can apply a different set of styles for when a page is printed than when it is displayed on a screen by using the `print` medium.

The `viewport` Meta Tag

For a site to look better on mobile devices, you need to add the `viewport` meta tag to your HTML, inside the `head` element:

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

This tells the browser how to set the page's dimensions and zoom level. The typical value used for width is `device-width`; however, this could also be set to an explicit pixel width.

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_12

Media Queries

A media query is defined as an at-rule, `@media`. It specifies a medium, such as `all`, `print`, `screen`, or `speech`, and a condition. You can omit the medium, in which case it defaults to `all`.

After the declaration, the media query has a block containing nested CSS rules. If the media query's condition is met, the CSS rules inside that block are applied to the document. If it is not met, the CSS rules are ignored.

Listing 12-1 has an example of a CSS rule, with a media query that overrides its style.

Listing 12-1. An example of a media query

```
h1 {
    color: blue;
}

@media screen and (max-width: 400px) {
    h1 {
        color: red;
    }
}
```

When the browser window has a width above 400 pixels, `h1` elements will be blue. Below 400 pixels, they will be red. Media queries are constantly being evaluated and styles conditionally applied – they aren't only calculated when the page first loads.

When the window is wide, the h1 elements will be blue. If you resize the window to a width of below 400 pixels, they will turn red. Widen it back above 400 pixels, and they will turn blue again.

With responsive design, the `min-width` and `max-width` rules are commonly used, as they are good indicators of viewport size. However, there are other media query rules available. For example, you can use the rule `orientation: landscape` to apply styles only when the user's device is in the landscape orientation.

Some queries are based on physical attributes of the device, such as width or orientation, but others are based on settings the user has configured in their operating system. One example of this is `prefers-reduced-motion`, which we discussed briefly in Chapter 9. Another example of a preference-based query is `prefers-color-scheme`. This can be used to detect if the user has configured their device for dark mode.

Logical Operators

Media queries also support multiple rules, with logical operators such as `and`, `or`, and `not`. For example:

```
@media screen
  and (min-width: 600px)
  and (orientation: landscape)
```

Media queries can also be used to conditionally load an entire stylesheet. For example, you may want to apply an entirely different stylesheet if the page is being printed, as Listing 12-2 demonstrates.

Listing 12-2. Conditionally loading stylesheets

```
<link rel="stylesheet" href="style.css" media="screen" />
<link rel="stylesheet" href="print.css" media="print" />
```

Range Syntax

You may want a media query to apply when a value falls within a certain range. For example, you might want to apply styles when the viewport width is between 100 and 300 pixels. You can accomplish this by using the range syntax:

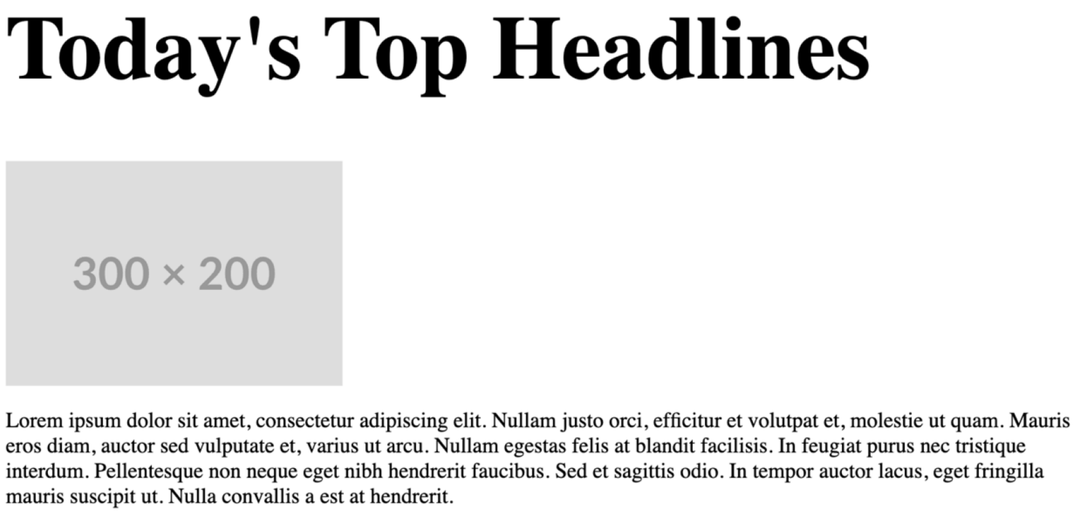
```
@media (100px <= width <= 300px)
```

Breakpoints

A breakpoint is the threshold at which a page's layout will change due to the viewport size with a media query. Defining your breakpoints is not an exact science. There are many different devices in use today, all with different screen sizes. For this reason, it's better to set breakpoints based on the content rather than targeting specific devices with media queries.

Instead, experiment with different viewport sizes, and find the points at which your layout and design start looking cramped. This will help you determine where to set your breakpoints.

Figure 12-1 shows an example layout, with a viewport width of 1,000 pixels.

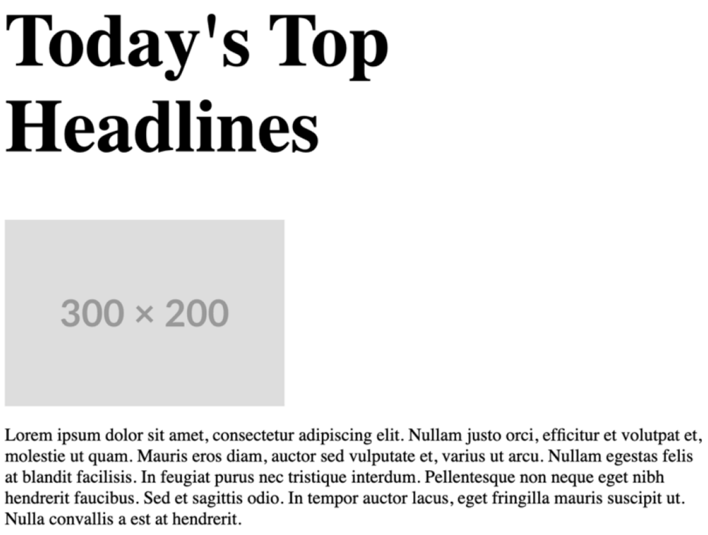

Today's Top Headlines

300 × 200

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-1. *An example layout*

If we start to shrink the viewport, the heading text wraps to two lines starting around a width of 800 pixels, as Figure 12-2 shows.


Today's Top
Headlines

300 × 200

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-2. *The heading text wrapped*

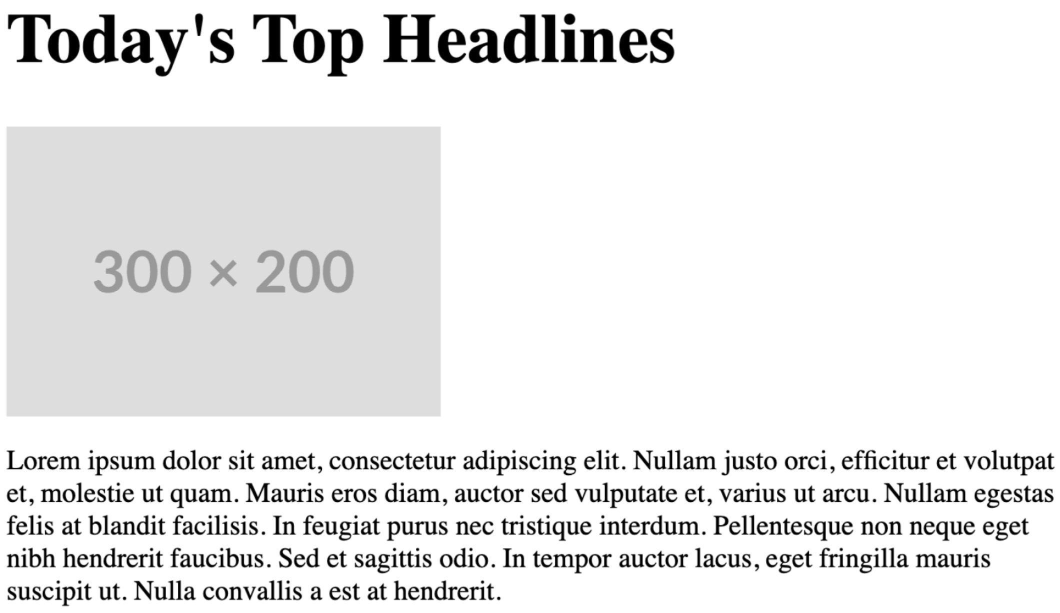
The headline takes up a lot of vertical space now, so this might be a good place for a breakpoint. We can make the font smaller when the viewport is 800 pixels or less by adding the media query shown in Listing 12-3.

Listing 12-3. Applying a media query to make the heading font smaller

```
h1 {
  font-size: 5rem;
}

@media (max-width: 800px) {
  h1 {
    font-size: 3rem;
  }
}
```

Now when we view this layout with a viewport width of less than 800 pixels, the heading font is smaller. It takes up less vertical space, since it doesn't wrap. Figure 12-3 shows the updated layout.

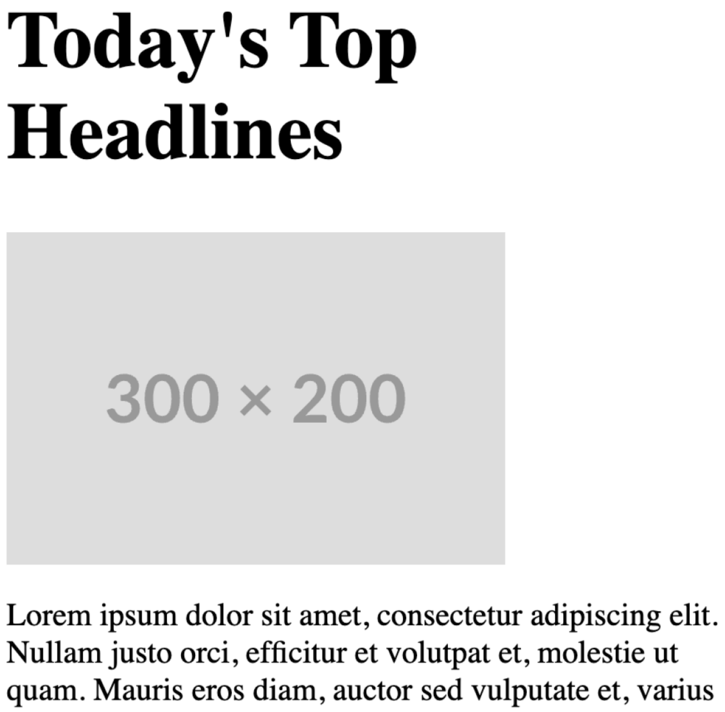


Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-3. *The heading text is smaller at this viewport size*

This will result in more content being visible on devices with a smaller viewport. If we make the width even smaller, we'll find that the heading wraps again at around 500 pixels, as Figure 12-4 shows.


Today's Top Headlines

300 × 200

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius

Figure 12-4. *The heading is wrapping again*

We can specify another breakpoint to again prevent the heading from wrapping. The new set of media queries is shown in Listing 12-4.

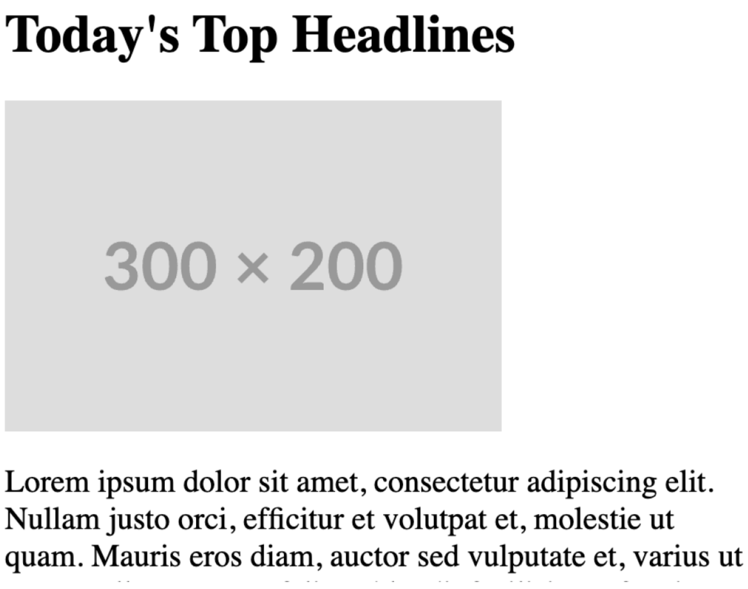
Listing 12-4. Adding another media query

```
h1 {
  font-size: 5rem;
}

@media (max-width: 800px) {
  h1 {
    font-size: 3rem;
  }
}
```

```
@media (max-width: 500px) {
  h1 {
    font-size: 2rem;
  }
}
```

Figure 12-5 shows the updated layout, with the smaller heading font size.



Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut

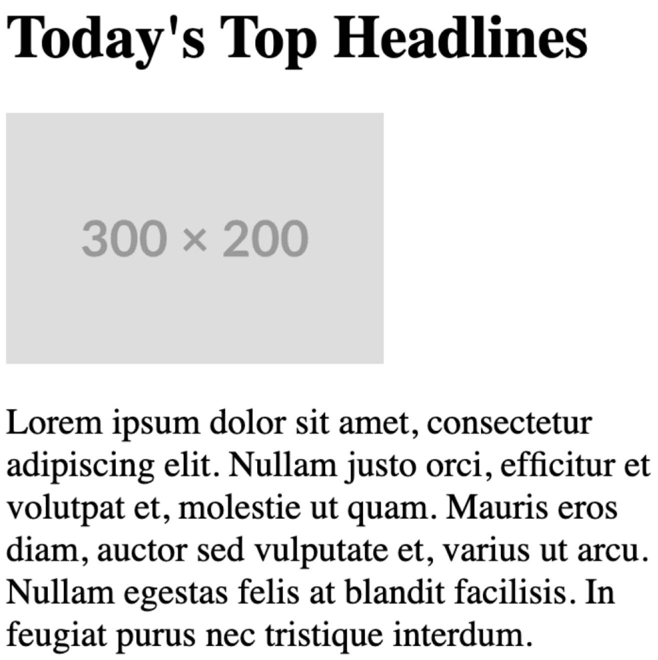
Figure 12-5. *The heading text gets even smaller at this viewport size*

At this viewport size, the image looks large compared to the headline size. We can also make the image a little smaller at this breakpoint, inside the same media query, as shown in Listing 12-5.

Listing 12-5. Adjusting the image size

```
@media (max-width: 500px) {
  h1 {
    font-size: 2rem;
  }
  img {
    width: 200px;
  }
}
```

With this change applied to the 500-pixel breakpoint, the image looks better sized relative to the rest of the content, as Figure 12-6 shows.



Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum.

Figure 12-6. *The layout with a smaller image*

Responsive Layouts with Flexbox

Some responsive layouts can be achieved without even using media queries. For example, we can use the `flex-wrap` property on a flex container to automatically wrap elements to the next line if the viewport is too narrow, to prevent the need for horizontal scrolling.

Listing 12-6 has an example layout that looks good on a wide viewport, but not so good on a narrow one.

Listing 12-6. An example layout

```
<style>
  .container {
    display: flex;
    gap: 8px;
    padding: 8px;
    background: #cccccc;
    justify-content: center;
  }

  .item {
    text-align: center;
    background: #eeeeee;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">Flex Item 1</div>
  <div class="item">Flex Item 2</div>
  <div class="item">Flex Item 3</div>
  <div class="item">Flex Item 4</div>
  <div class="item">Flex Item 5</div>
  <div class="item">Flex Item 6</div>
</div>
```

With a wide viewport, this layout looks good, as Figure 12-7 shows.

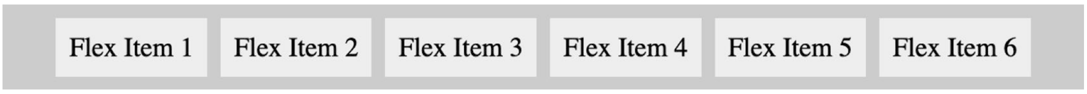


Figure 12-7. *The rendered layout with a wide viewport*

With a narrower width, the layout looks a little cramped and the text inside is wrapped, as Figure 12-8 shows.

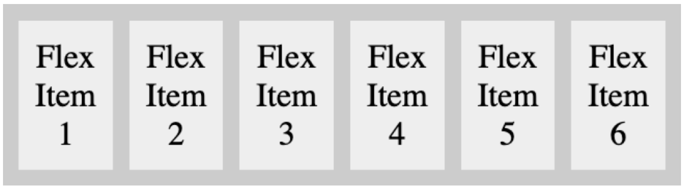


Figure 12-8. *The layout with a narrow viewport*

Let's go even narrower, shown in Figure 12-9. The items can't shrink any further, and they overflow the flex container.

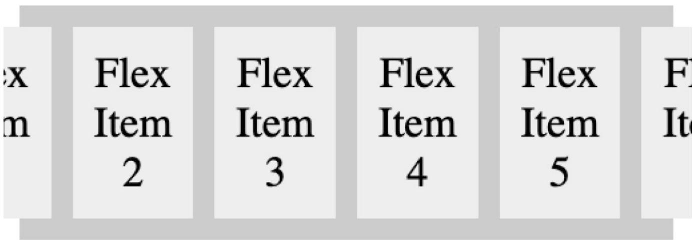


Figure 12-9. *The flex items overflowing the container*

We can make this layout responsive by setting the `flex-wrap` property to `wrap` on the container. Listing 12-7 has the updated layout code.

Listing 12-7. Making the flex layout responsive

```
<style>
  .container {
    display: flex;
    flex-wrap: wrap;
    gap: 8px;
    padding: 8px;
    background: #cccccc;
    justify-content: center;
  }

  .item {
    text-align: center;
    background: #eeeeee;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">Flex Item 1</div>
  <div class="item">Flex Item 2</div>
  <div class="item">Flex Item 3</div>
  <div class="item">Flex Item 4</div>
  <div class="item">Flex Item 5</div>
  <div class="item">Flex Item 6</div>
</div>
```

Figure 12-10 shows this new layout with a narrow viewport.

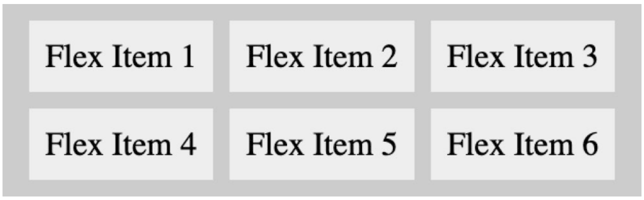


Figure 12-10. *The wrapped layout*

This is much better. The items are more readable. If we reduce the viewport width even further, the layout will change to accommodate the new size, as Figure 12-11 shows.

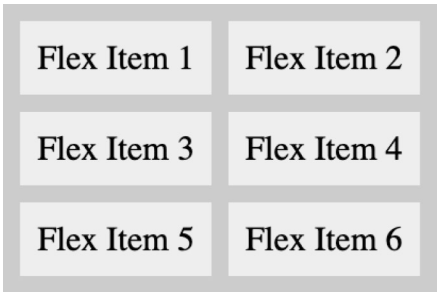


Figure 12-11. *The layout wrapping changes to fit the narrow viewport*

Responsive Layouts with CSS Grid

Grid layouts can also be made responsive. There are a few different ways you can do this.

Using `auto-fit`

In Chapter 11, we saw the `auto-fit` keyword for grid column sizing. Listing 12-8 shows how this can be used to make the layout more responsive.

Listing 12-8. A responsive grid layout with `auto-fit`

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
    gap: 8px;
    padding: 8px;
    background: #cccccc;
  }

  .item {
    background: #eeeeee;
    text-align: center;
    padding: 8px;
  }
</style>

<div class="container">
  <div class="item">Grid Item 1</div>
  <div class="item">Grid Item 2</div>
  <div class="item">Grid Item 3</div>
  <div class="item">Grid Item 4</div>
  <div class="item">Grid Item 5</div>
  <div class="item">Grid Item 6</div>
</div>
```

With a viewport width of 600 pixels, the grid is rendered with two rows and three columns, as Figure 12-12 shows.

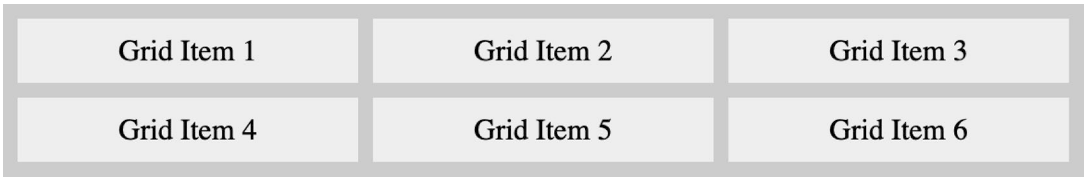


Figure 12-12. *A 2x3 grid layout*

If we resize the viewport down to 400 pixels, the grid columns are automatically adapted so that the items fit, as Figure 12-13 shows.

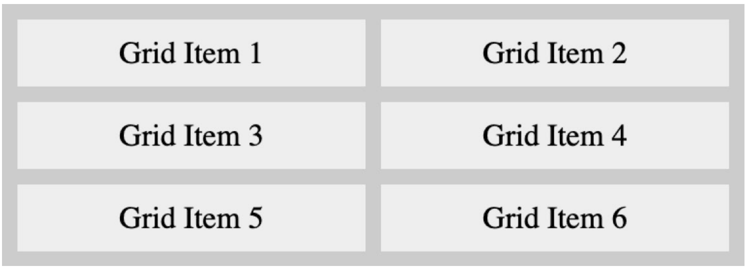


Figure 12-13. *The resized grid layout*

Changing the Grid Layout with a Media Query

If you want more control over how the layout adapts to different viewport widths, you can use a media query. An example of this is shown in Listing 12-9.

Listing 12-9. A responsive grid layout

```
<style>
  .container {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
    background: #cccccc;
    gap: 8px;
    padding: 8px;
  }
  .item {
    padding: 8px;
    text-align: center;
    background: #eeeeee;
  }

  @media (max-width: 350px) {
    .container {
      grid-template-columns: 1fr;
    }
  }
</style>

<div class="container">
  <div class="item">Grid Item 1</div>
  <div class="item">Grid Item 2</div>
  <div class="item">Grid Item 3</div>
  <div class="item">Grid Item 4</div>
  <div class="item">Grid Item 5</div>
  <div class="item">Grid Item 6</div>
</div>
```

Figure 12-14 shows the layout with a 600-pixel viewport. There are two rows and three columns.

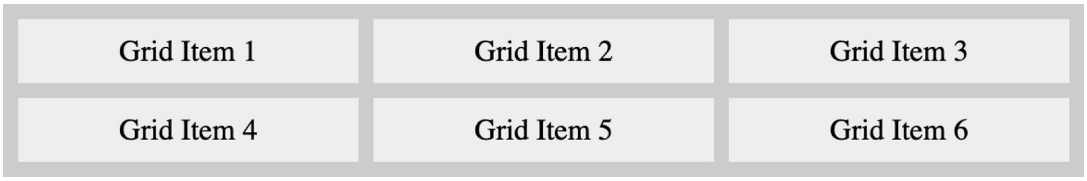


Figure 12-14. *The two-row grid layout*

If we resize the viewport to below 350 pixels, the new `grid-template-columns` from the media query takes effect, changing the grid layout to have a single column, as Figure 12-15 shows.

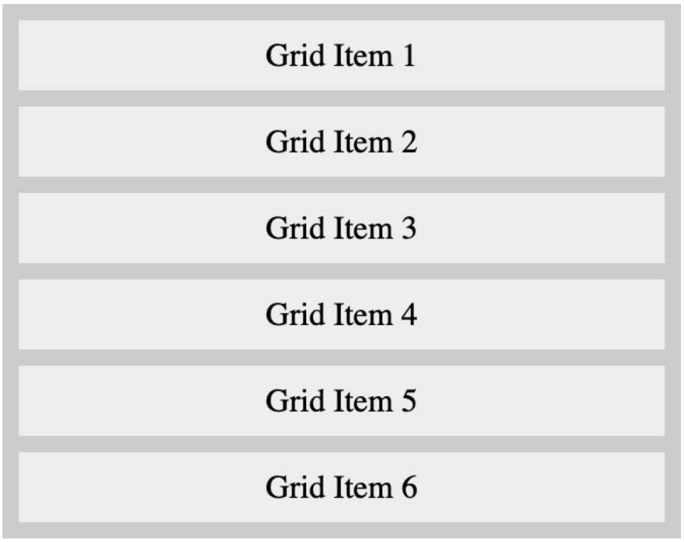
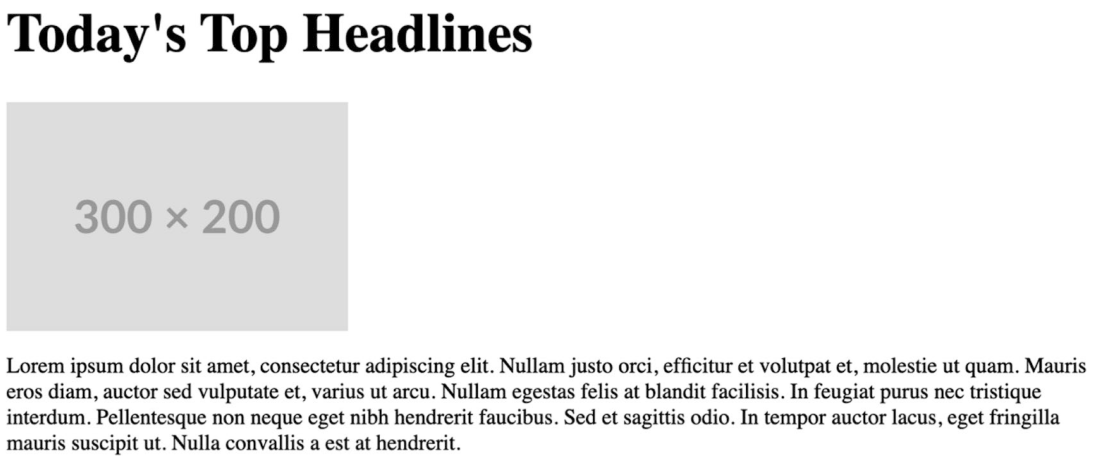


Figure 12-15. *The single-column grid layout*

Fluid Typography

Earlier, we saw how to leverage media queries to adjust the font size as the viewport size changes. With fluid typography, it's possible to automatically scale the font size to viewport size without having to use media queries.

In Chapter 3, we discussed the `vw` unit, which is equal to 1% of the viewport width. We can use `vw` to specify a font size as a proportion of the viewport width. Suppose that you want your header text to have a size of 48 pixels when the viewport is 1,000 pixels wide. 48 is 4.8% of 1,000, so you can use a font size of `4.8vw`. Figure 12-16 shows what that might look like.

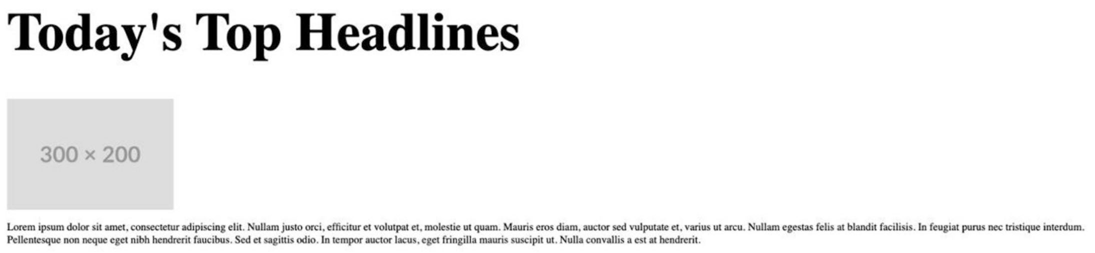


Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-16. *The layout with a 1,000-pixel width*

There's a slight problem with this approach. If the viewport becomes very wide, the font size of the heading scales accordingly to a very large size. For example, at a viewport size of 2,000 pixels, the font size becomes `96px`. Notice the very large font in Figure 12-17 compared to the body text size.

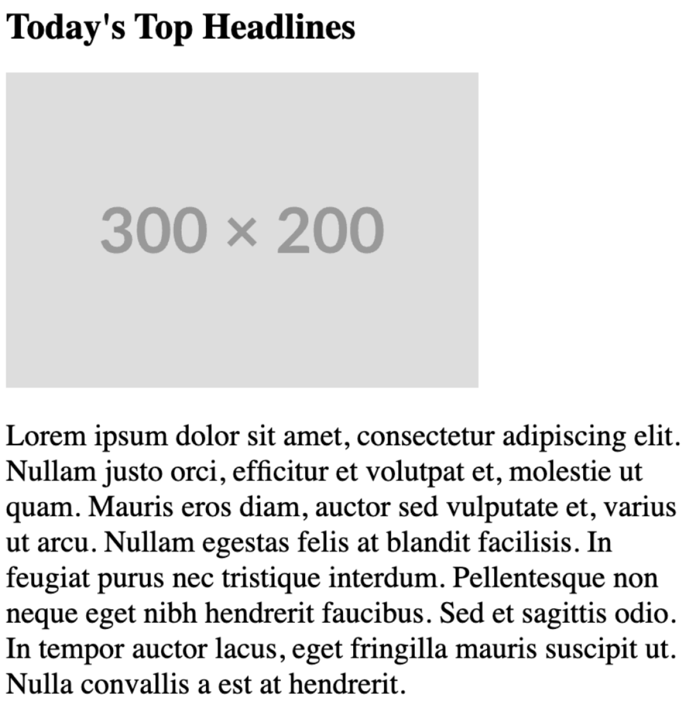


Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-17. *A very large font size*

On the other hand, if the viewport becomes very narrow, the heading's font size becomes much smaller – almost as small as the body text. Figure 12-18 shows this layout with a viewport width of 480 pixels.


Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-18. *The heading font becomes very small.*

Limiting the Font Size with the `clamp` Function

We can limit how large or small the header font gets using the `clamp` function. This function takes three arguments:

- The minimum size
- The preferred size
- The maximum size

We can keep the font size of `4.8vw` as the preferred size and establish a lower and upper bound. With a lower bound of 48px and an upper bound of 64px, we can set the font size as shown in Listing 12-10.

Listing 12-10. Using the `clamp` function

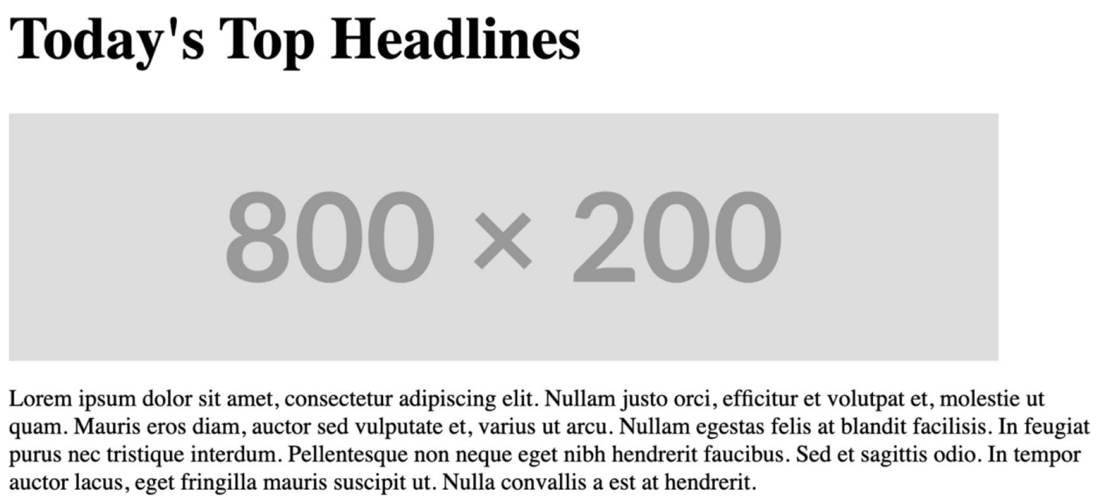
```
h1 {
  font-size: clamp(48px, 4.8vw, 64px);
}
```

Now the font size adjusts automatically with the viewport width as before but will never be smaller than 48 pixels or larger than 64 pixels.

Making Images Responsive

Earlier, we used a media query to change the size of an image if the viewport was narrower than a certain threshold. This works but is not very flexible. There is another technique that we can use that allows images to scale with the viewport width without using media queries.

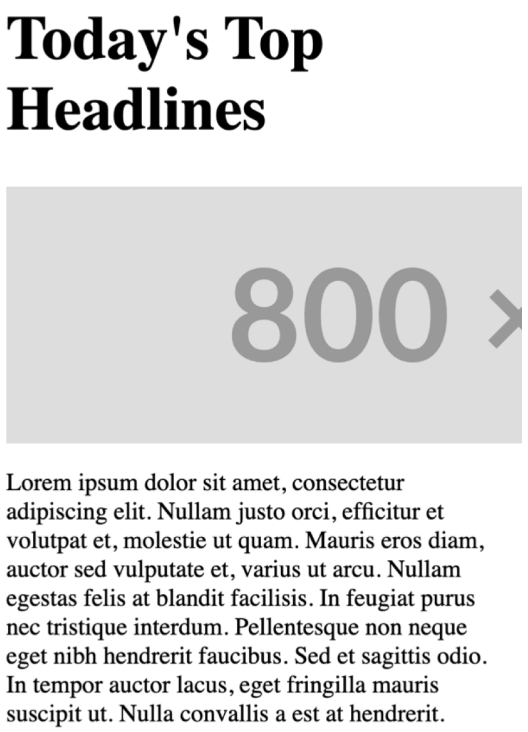
Figure 12-19 shows the layout from the previous section, using an image that is 800 pixels wide.


Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-19. *The layout with a large image*

If the viewport is resized below 800 pixels wide, the image is cut off and requires horizontal scrolling to view the rest of it, as Figure 12-20 shows.


Today's Top Headlines

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque eget nibh hendrerit faucibus. Sed et sagittis odio. In tempor auctor lacus, eget fringilla mauris suscipit ut. Nulla convallis a est at hendrerit.

Figure 12-20. *The image is cut off with a narrow viewport*

To fix this, you can change the image sizing behavior so that the image resizes along with the viewport with two CSS changes, shown in Listing 12-11.

Listing 12-11. Making the image responsive

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

This sizes the image so that it takes up the full width of its container. The `height: auto` makes sure that the image retains its aspect ratio.

With this change, the image is resized along with the viewport and is fully visible even at a narrow width, as Figure 12-21 shows.

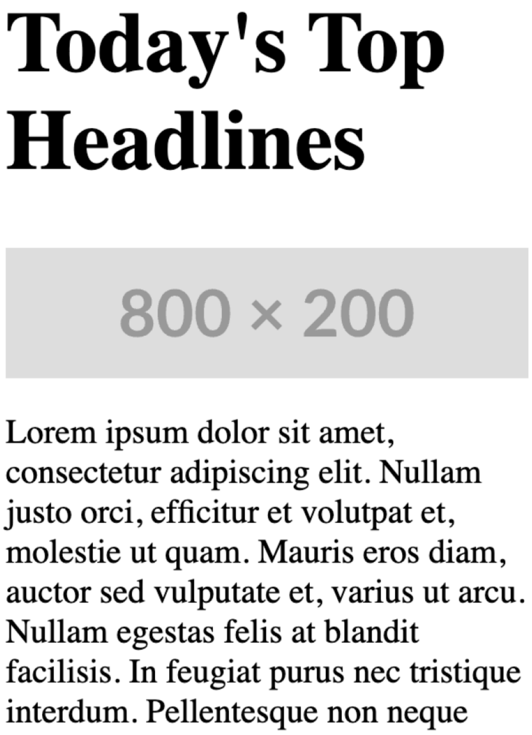

Today's Top Headlines
800 × 200
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam justo orci, efficitur et volutpat et, molestie ut quam. Mauris eros diam, auctor sed vulputate et, varius ut arcu. Nullam egestas felis at blandit facilisis. In feugiat purus nec tristique interdum. Pellentesque non neque

Figure 12-21. *The image now resizes with the viewport*

Complete Example: Responsive Page Layout

In this section, we'll take a full-page layout and improve its responsiveness for small screen sizes. Listing 12-12 has the initial page layout.

Listing 12-12. The initial page layout

```
body {
  margin: 0;
}

.container {
  display: flex;
  flex-direction: column;
  height: 100vh;
}
```

```
.header {
  background: #64748b;
  color: #ffffff;
  padding: 16px;
}

.main {
  display: flex;
  flex-direction: row;
  flex-grow: 1;
}

.content {
  background: #fafaf9;
  padding: 1rem;
  flex-grow: 1;
}

.sidebar {
  display: flex;
  flex-direction: column;
  gap: 16px;
  background: #cbd5e1;
  padding: 16px;
}

.sidebar a {
  padding: 8px 16px;
  background: #94a3b8;
  color: #000000;
  text-decoration: none;
}

.sidebar2 {
  background: #cbd5e1;
  padding: 16px;
}
```

```
  .footer {
    background: #334155;
    color: #ffffff;
    padding: 16px;
  }
</style>

<div class="container">
  <header class="header">Header</header>
  <main class="main">
    <nav class="sidebar">
      <a href="/home">Home</a>
      <a href="/about">About</a>
      <a href="/photos">Photos</a>
    </nav>
    <div class="content">Hello world!</div>
    <div class="sidebar2">Sidebar 2</div>
  </main>
  <footer class="footer">Footer</footer>
</div>
```

Figure 12-22 shows the initial rendered layout.

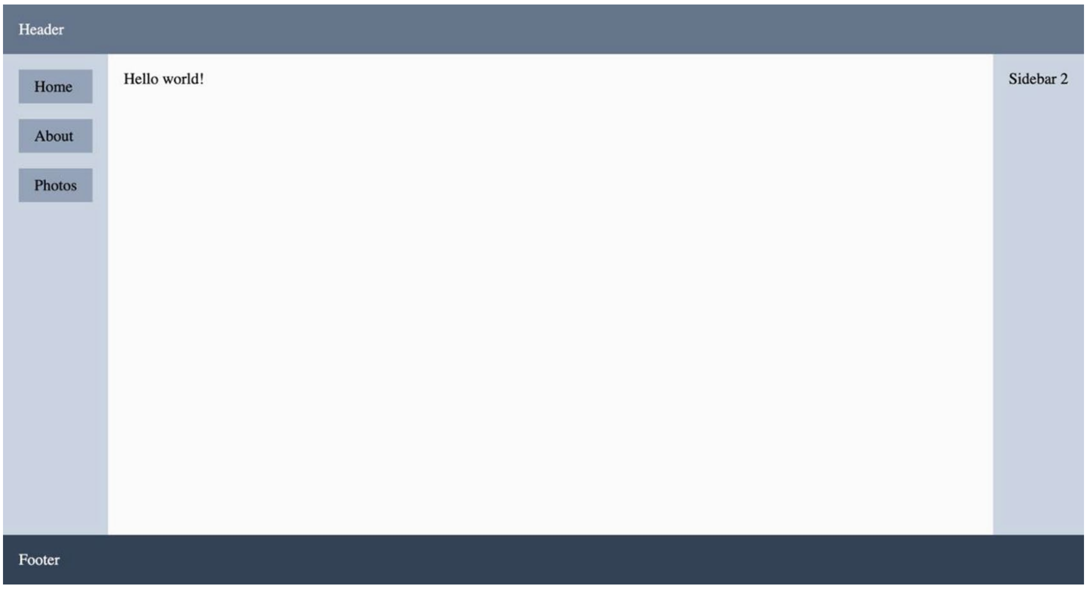


Figure 12-22. *The initial page layout*

This looks good on a wide desktop layout, but how does this look on a mobile device? If we decrease the viewport width to 402 pixels (the number of logical pixels in the iPhone 16 Pro), everything's still visible, but the layout is very narrow. Figure 12-23 shows the mobile layout.

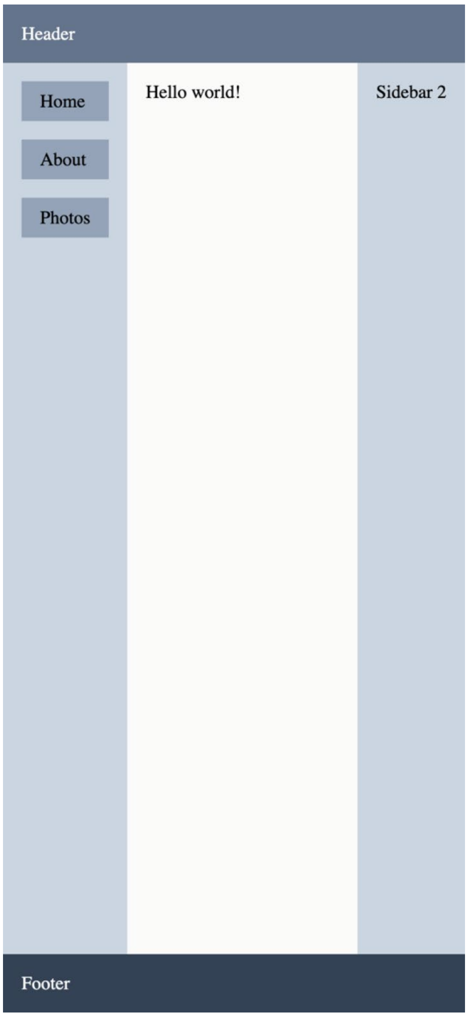


Figure 12-23. *The mobile view of this layout*

This is very cramped. There's not much room for the main or sidebar content. This isn't ideal for a mobile device viewing our layout. We can improve this by adding a media query and making a few layout changes.

You can experiment a bit to find a good breakpoint for the media query, but to make sure the content area never gets too narrow, let's set the breakpoint at 500 pixels. Figure 12-24 shows the layout at a width of 500 pixels.

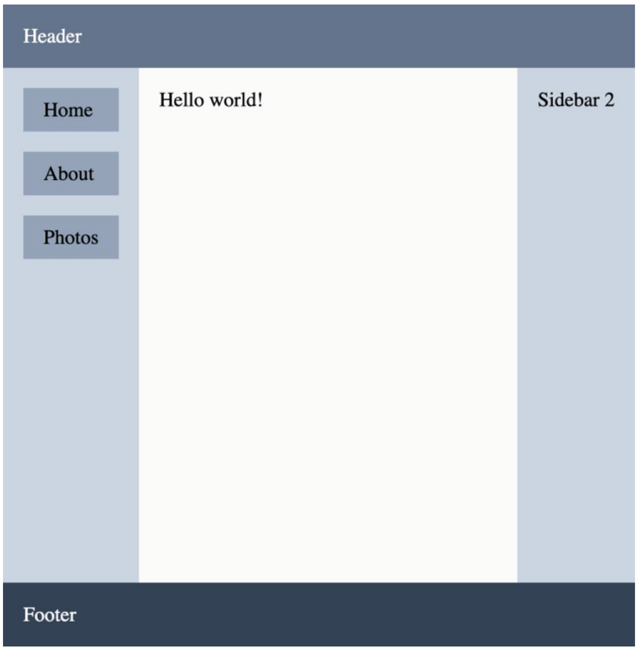


Figure 12-24. *The layout at the 500-pixel breakpoint*

Below this threshold, we'll change the `main` element to have a `flex-direction` of `column`. This will stack the three regions vertically, allowing them each to take up the full screen width. Listing 12-13 shows the media query and its associated CSS rule.

Listing 12-13. Adding a media query for the narrow viewport

```
@media screen and (max-width: 500px) {
  .main {
    flex-direction: column;
  }
}
```

Now the content takes up the full width and is no longer constrained horizontally. The new layout takes effect at 500 pixels or less. Figure 12-25 shows the new layout.

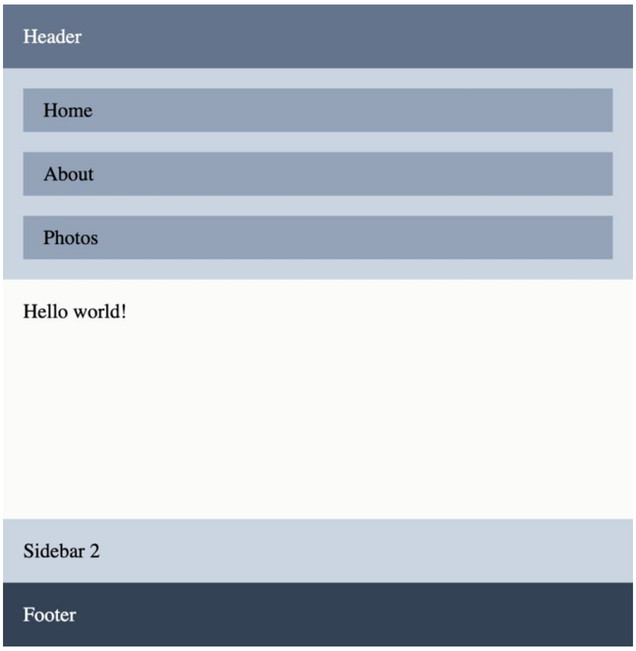


Figure 12-25. *The adjusted layout at its breakpoint of 500 pixels*

This is good, but it could be better. The navigation is taking up a lot of vertical space now. Listing 12-14 adds another rule inside the 500-pixel media query to change the navigation links to be horizontal.

Listing 12-14. Adapting the navigation to the narrow view

```css
@media screen and (max-width: 500px) {
  .main {
    flex-direction: column;
  }

  .sidebar {
    flex-direction: row;
    justify-content: center;
  }
}
```

Figure 12-26 shows the updated layout.

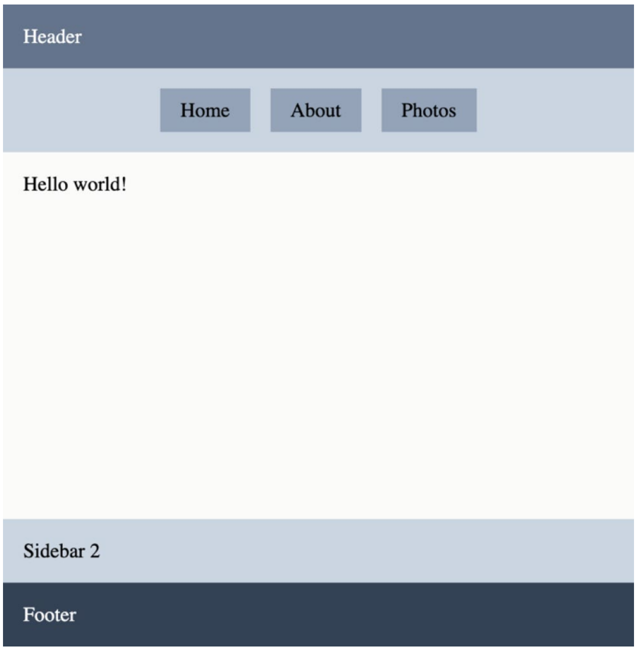


Figure 12-26. *The navigation links laid out horizontally*

This responsive layout looks a lot better with a viewport width of 500 pixels. There is still a minor issue with the layout, though. Let's add a few more navigation items and reduce the width to 400 pixels. Figure 12-27 shows the result.

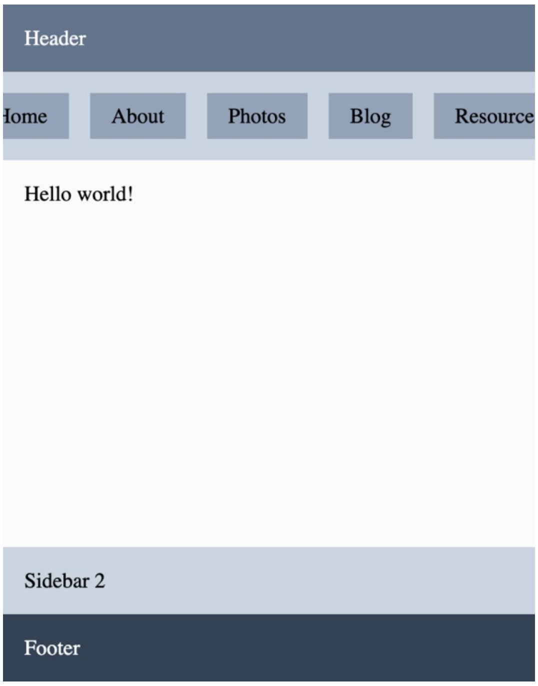


Figure 12-27. *Adding navigation links and narrowing the viewport*

With the reduced width and additional navigation items, the navigation now overflows the viewport. We can fix this by adding `flex-wrap: wrap` to the navigation container, as Listing 12-15 shows.

Listing 12-15. Improving the responsiveness of the navigation items

```
@media screen and (max-width: 500px) {
  .main {
    flex-direction: column;
  }

  .sidebar {
    flex-direction: row;
    justify-content: center;
    flex-wrap: wrap;
  }
}
```

As Figure 12-28 shows, the navigation items no longer overflow the viewport. Allowing the container to wrap lets us fit all the navigation items in the available width.

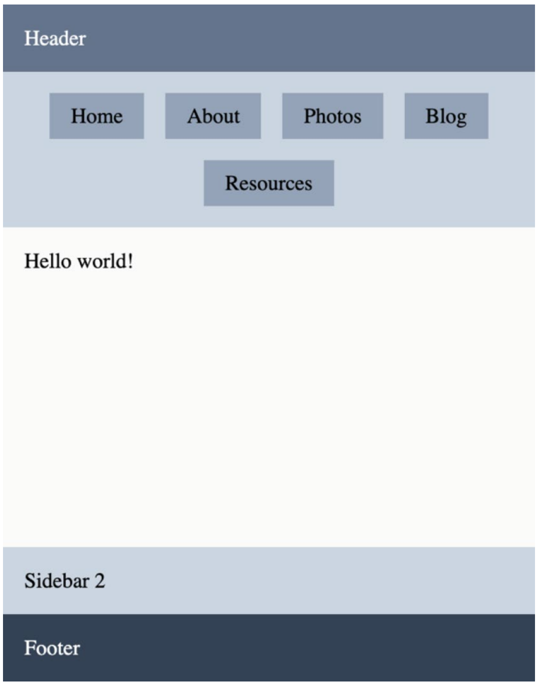


Figure 12-28. *The navigation items now wrap*

Container Size Queries

Media queries are a versatile tool for creating responsive layouts, but they have a limitation. They can only adapt the layout based on the viewport width. Container size queries let you adapt an element's layout based on the size of its container element.

Listing 12-16 has a two-column layout, showing some metadata about a user. The left column is only 100 pixels wide, and the metadata content wraps and overflows.

Listing 12-16. A two-column metadata layout

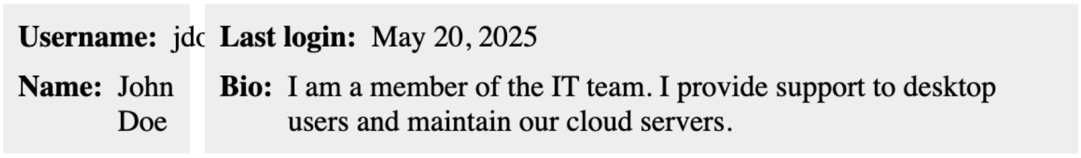
```
<style>
  .container {
    display: grid;
    grid-template-columns: 100px 1fr;
    gap: 8px;
    padding: 8px;
  }

  .container > div {
    background: #eeeeee;
    display: flex;
    flex-direction: column;
    gap: 8px;
    padding: 8px;
  }

  .metadata {
    display: flex;
    gap: 8px;
  }
</style>

<div class="container">
  <div>
    <div class="metadata">
      <strong>Username:</strong>
      <div>jdoe</div>
    </div>
    <div class="metadata">
      <strong>Name:</strong>
      <div>John Doe</div>
    </div>
  </div>
  <div>
    <div class="metadata">
      <strong>Last login:</strong>
      <div>May 20, 2025</div>
    </div>
    <div class="metadata">
      <strong>Bio:</strong>
      <div>
        I am a member of the IT team. I provide
        support to desktop users and maintain our cloud servers.
      </div>
    </div>
  </div>
</div>
```

You can see the resulting layout, with its wrapping and overflowing content, in Figure 12-29.



Username: jdc **Last login:** May 20, 2025

Name: John Doe **Bio:** I am a member of the IT team. I provide support to desktop users and maintain our cloud servers.

Figure 12-29. *The rendered layout*

The left column will be narrow regardless of the viewport size, so a media query won't help here. We can use a container query to adapt the flex layout of metadata items so that they use a `flex-direction` of `column` when the container is narrow.

To use a container query, an element's container must be declared to be a query container element. Otherwise, the container query has no effect. This is done with the `container-type` property. By default, this is set to `normal`, which means it will not work with container size queries.

To use a container size query based on the container's width, we can set container-type to `inline-size`. This means to use the size of the element along the inline axis (the width). With our layout, we want to apply this property to each grid item (the immediate `div` child elements under the grid container). Listing 12-17 adds the `container-type` property to this CSS rule.

Listing 12-17. Setting a container type

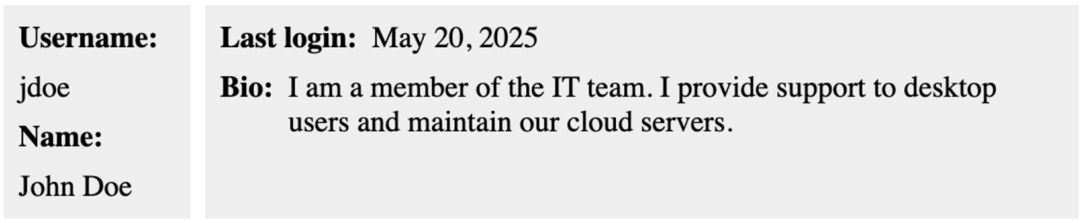
```
.container > div {
  background: #eeeeee;
  display: flex;
  flex-direction: column;
  gap: 8px;
  padding: 8px;
  container-type: inline-size;
}
```

Now that we've set a container type, we can add a container size query in Listing 12-18.

Listing 12-18. Adding a container query to adapt the metadata layout

```
@container (max-width: 300px) {
  .metadata {
    flex-direction: column;
  }
}
```

Figure 12-30 shows the updated layout. Now, when a metadata element is in a container that's narrower than 300 pixels, we set `flex-direction` to `column` so the metadata labels and values are stacked vertically.



Username:
jdoe
Name:
John Doe

Last login: May 20, 2025
Bio: I am a member of the IT team. I provide support to desktop users and maintain our cloud servers.

Figure 12-30. *The layout adapted with a container size query*

A container query works on the closest ancestor element with `container-type` set. If you want more fine-grained control over the container, you can create a named containment context. Let's adapt the previous layout to use a named containment context.

Listing 12-19 assigns a name with the `container-name` property.

Listing 12-19. Creating a named containment context

```
.container > div {
  background: #eeeeee;
  display: flex;
  flex-direction: column;
  gap: 8px;
  padding: 8px;
  container-type: inline-size;
  container-name: metadataContainer;
}
```

Next, we add the context name to the container size query in Listing 12-20.

Listing 12-20. Using the named containment context

```
@container metadataContainer (max-width: 300px) {
  .metadata {
    flex-direction: column;
  }
}
```

This yields the same result as shown in Figure 12-30. In this case, though, the container query will only take effect for an ancestor element with a `container-name` of `metadataContainer`.

Container Query Units

Like the `vw` and `vh` units, which are proportional to the viewport size, you can use container query units that are proportional to the query container size. These units include

- `cqw`: 1% of the container's width
- `cqh`: 1% of the container's height

Instead of width and height, you can use container query units based on logical properties. For a horizontal writing mode, these correspond to width and height, respectively:

- `cqi`: 1% of the container's inline size
- `cqb`: 1% of the container's block size

Summary

- Media queries let you conditionally apply styles based on, among other things, the size of the viewport.
- A breakpoint is a particular width at which your layout changes to be more responsive.
- A flexbox layout can be made more responsive by setting `flex-wrap` to `wrap`.
- A grid layout can be made more responsive by using `auto-fit`.
- Text can be sized based on the viewport size with the `vw` and `vh` units.
- Text size can be limited with an upper and lower bound using the `clamp` function.
- An image can be made more responsive by setting `max-width` to `100%` and `height` to `auto`.
- Container size queries let you conditionally apply styles based on the size of an ancestor container element.

CHAPTER 13

Wrap Up

Before we end our exploration of modern CSS, let's talk about a few miscellaneous CSS features that are worth your attention.

Limiting the Scope of CSS Rules with @scope

CSS selectors match elements across the entire document. This means you need unique class names or very specific selectors to limit the scope of your style rules. The @scope at-rule lets you narrow the scope of your CSS selectors.

You can specify a selector for the scope, and any rules contained inside the @scope rule will only apply to elements that are descendants of matching elements. Listing 13-1 has an example of using @scope to style elements with the same class in different ways.

Listing 13-1. Using the @scope rule

```
<style>
    .container {
        display: flex;
        gap: 8px;
        margin: 8px;
    }

    .box {
        width: 64px;
        height: 64px;
        background: blue;
    }
```

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4_13

```
    @scope (.red-boxes) {
        .box {
            background: red;
        }
    }
</style>

<div class="container">
    <div class="box"></div>
    <div class="box"></div>
</div>

<div class="container red-boxes">
    <div class="box"></div>
    <div class="box"></div>
</div>
```

Figure 13-1 shows the result.

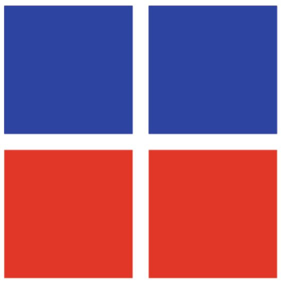


***Figure 13-1.** An example of using scoped styles*

All four boxes are styled by the first `.box` rule, which sets the width and height and color. There's also a @scope rule that applies a different `.box` rule that only applies to descendants of elements matching the `.red-boxes` selector.

Note This rule may not be supported in all browsers at the time you're reading this.

Reading an Element's Attributes

The CSS attr function lets you read an element's attributes from a CSS rule. This attribute can then be used as part of a CSS property. Listing 13-2 shows how you can use this function to style a elements to include the full URL they link to.

Listing 13-2. Using the attr function

```
<style>
    a::after {
        content: " (" attr(href) ")"
    }
</style>
```

```
<a href="https://google.com">Google</a>
```

The result is shown in Figure 13-2.

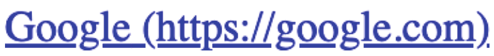


Figure 13-2. *The styled link*

We're using the ::after pseudo-element to add the value of the link's href attribute in parentheses after the link text.

Scroll Snapping

When you add scrolling to a container element by setting overflow to auto, the content scrolls smoothly by default. You can change the scrolling behavior so that it "snaps" to child elements inside the scroll container.

If you scroll partially to a child element and stop scrolling, the scrolling viewport will snap to the configured position.

Configuring Scroll Snap on the Container

Scroll snapping can be enabled in a few different ways on the container element with the scroll-snap-type property. You can snap in the X and/or Y directions, with one of two policies. This can be set to be mandatory, which means the content will always snap. It can also be proximity, which means the browser can decide whether to snap.

These two values are passed to the scroll-snap-type property, separated by a space. See Listing 13-3 for an example.

Configuring Scroll Snap on Children

The second part of enabling scroll snap is setting the alignment point when the scroll position snaps with the `scroll-snap-align` property. This can be set to `center`, `start`, or `end`.

Listing 13-3 has a full example of scroll snapping.

Listing 13-3. Applying scroll snap behavior

```
<style>
  .container {
    display: flex;
    gap: 8px;
    width: 110px;
    overflow: auto;
    scroll-snap-type: x mandatory;
  }

  .box {
    background: #cccccc;
    text-align: center;
    padding: 50px;
    scroll-snap-align: start;
  }
</style>

<div class="container">
  <div class="box">1</div>
  <div class="box">2</div>
  <div class="box">3</div>
</div>
```

Checking for Feature Support with `@supports`

You can apply CSS rules conditionally based on the browser's support for certain features. For example, Listing 13-4 shows how you can use `@supports` to check if the browser supports CSS Grid.

Listing 13-4. Checking for support with the @supports rule

```
<style>
  @supports (display: grid) {
    .container {
      display: grid;
      grid-template-columns: 1fr 1fr;
    }
  }
</style>

<div class="container">
  <div>1</div>
  <div>2</div>
</div>
```

The styles inside the `@supports` rule will only be applied if the browser supports `display: grid`. If the feature is not supported, all the rules inside the block will be ignored.

You can also use `@supports` to provide fallback styles for when the browser does *not* support a given feature by using the `not` keyword. Listing 13-5 shows an example of using flexbox as a fallback if CSS Grid isn't supported.

Listing 13-5. Providing fallback styles

```
<style>
  @supports (display: grid) {
    .container {
      display: grid;
      grid-template-columns: 1fr 1fr;
    }
  }
  @supports (not (display: grid)) {
    .container {
      display: flex;
      gap: 8px;
    }
  }
</style>

<div class="container">
  <div>1</div>
  <div>2</div>
</div>
```

Applying Visual Effects with the filter Property

The `filter` property lets you apply visual effects to elements. You can apply a blur effect or modify the colors of an element, depending on the value of this property. The values of this property are function calls. Consider the image shown in Figure 13-3.

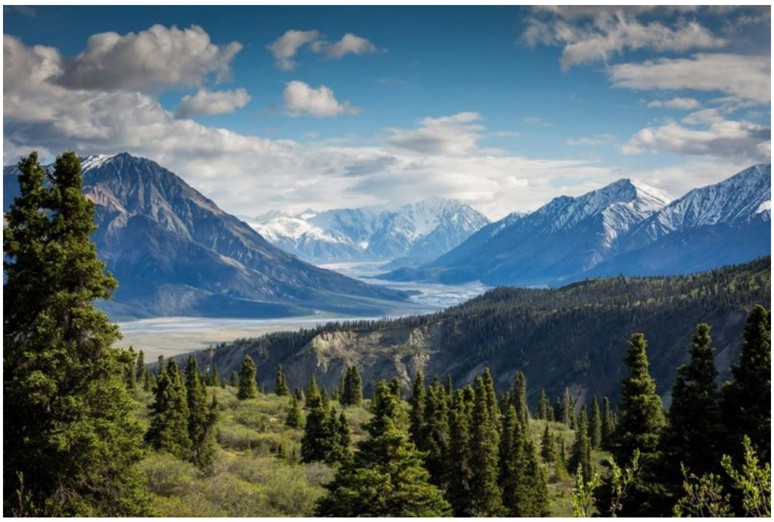


Figure 13-3. *The unfiltered image*

You can apply a blur effect by setting the `filter` property to `blur(10px)`, producing the result shown in Figure 13-4.

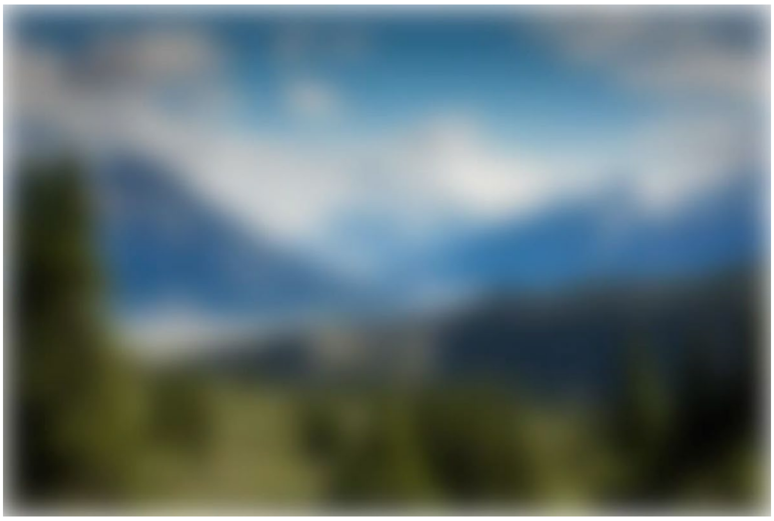


Figure 13-4. *The blurred image*

You can use a filter of `grayscale(100%)` to turn the image fully grayscale, as shown in Figure 13-5.

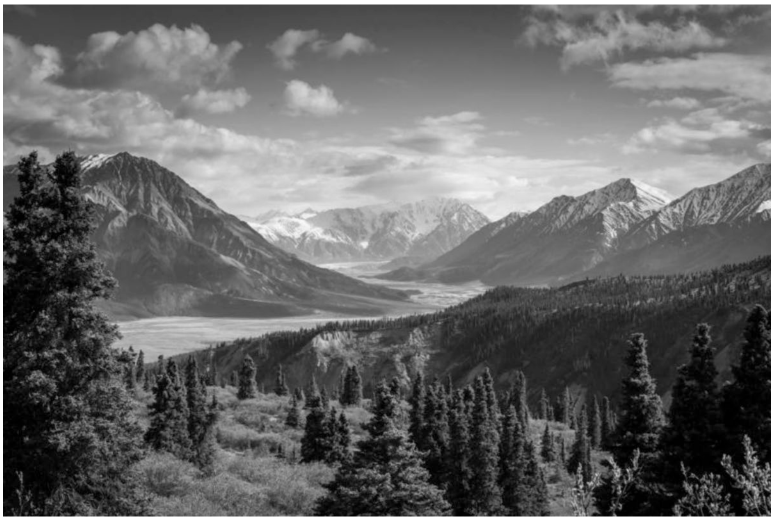


Figure 13-5. *The grayscale image*

There are several other filters you can apply, such as `brightness`, `contrast`, `invert`, and more.

Mixing Colors with color-mix

CSS allows you to combine two colors, mixing them at a certain ratio. This lets you create colors derived from other colors. Listing 13-6 has an example of mixing red and blue in the HSL color space.

Listing 13-6. Mixing two HSL colors

```
<style>
  .box {
    width: 100px;
    height: 100px;
    background: color-mix(in hsl, red 50%, blue 50%);
  }
</style>

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

Figure 13-6 shows the resulting color.

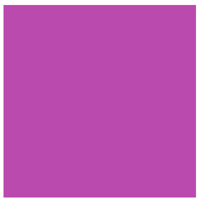


Figure 13-6. *The mixed color*

You can mix a color with white to make a lighter shade or with black to make a darker shade. Listing 13-7 shows how to lighten and darken a color using this technique.

Listing 13-7. Creating different shades of a base color

```
<style>
  .container {
    display: flex;
    gap: 8px;
  }
  .box {
    width: 100px;
    height: 100px;
  }

  .base {
    background: red;
  }

  .lighter {
    background: color-mix(in hsl, red 50%, white 50%);
  }

  .darker {
    background: color-mix(in hsl, red 50%, black 50%);
  }
</style>

<div class="container">
  <div class="box lighter"></div>
  <div class="box base"></div>
  <div class="box darker"></div>
</div>
```

Figure 13-7 shows the resulting colors.

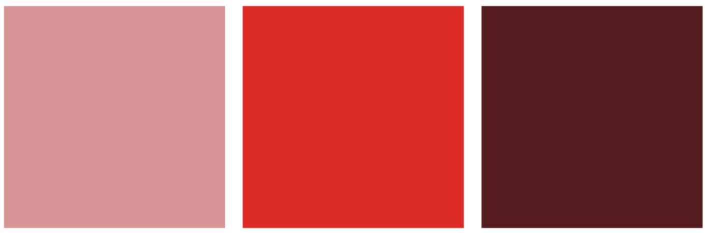


Figure 13-7. *The different shades of red*

You can also use `color-mix` with CSS variables. In Listing 13-8, we define a variable for the button color and use `color-mix` to create a lightened version of that color for when the button is hovered.

Listing 13-8. Using CSS variables with `color-mix`

```
<style>
  :root {
    --button-background: hsl(201 96 32);
  }

  button {
    padding: 8px 16px;
    border: none;
    color: white;
    background: var(--button-background);
  }

  button:hover {
    background:
        color-mix(in hsl, var(--button-background) 100%, white 25%);
  }
</style>

<button>Hover me!</button>
```

Utility-First CSS

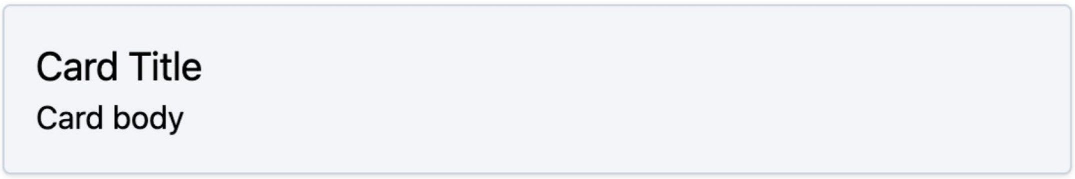
Utility-first CSS frameworks are continuing to gain popularity. These frameworks are a very different way of thinking about CSS. Instead of writing your own CSS rules that apply styles to elements, there are predefined "utility" classes that apply different types of styling. Generally, there is one class name per CSS property and value combination.

Tailwind CSS is the most popular and widely used utility-first CSS framework. Listing 13-9 shows an example of a simple "card" component using Tailwind CSS classes.

Listing 13-9. A card component with Tailwind CSS

```
<div class="rounded border border-slate-300 bg-slate-100 p-4 shadow">
  <h2 class="text-xl">Card Title</h2>
  <p>Card body</p>
</div>
```

This results in the card component shown in Figure 13-8.


Card Title

Card body

Figure 13-8. *The card component using Tailwind styles*

Each class name applies a separate piece of styling. For example:

- `p-4` applies padding.
- `bg-slate-100` applies a background color.
- `shadow` applies a box shadow.
- `rounded` applies a border radius.
- `border border-slate-300` applies a solid, colored border.

Final Words

As you have seen, there have been many new and exciting developments in the world of CSS since the first edition of this book. There are many more on the way that will give you, the web developer, more power and control over styling your websites and apps.

I hope this book has helped you to chart your journey into further exploration of CSS.

Index

© Joe Attardi 2025
J. Attardi, *Modern CSS*, https://doi.org/10.1007/979-8-8688-1727-4