Table of Contents

[Cascade 5](#_Toc143076884)

[Understanding stylesheet origin 5](#_Toc143076885)

[USER AGENT STYLES 5](#_Toc143076886)

[IMPORTANT DECLARATIONS 5](#_Toc143076887)

[Understanding specificity 6](#_Toc143076888)

[INLINE STYLES 6](#_Toc143076889)

[SELECTOR SPECIFICITY 6](#_Toc143076890)

[A NOTATION FOR SPECIFICITY 6](#_Toc143076891)

[SPECIFICITY CONSIDERATIONS 6](#_Toc143076892)

[Understanding source order 6](#_Toc143076893)

[LINK STYLES AND SOURCE ORDER 7](#_Toc143076894)

[CASCADED VALUES 7](#_Toc143076895)

[Two rules of thumb 7](#_Toc143076896)

[Inheritance 7](#_Toc143076897)

[Special values 7](#_Toc143076898)

[Using the inherit keyword 7](#_Toc143076899)

[Using the initial keyword 7](#_Toc143076900)

[Shorthand properties 8](#_Toc143076901)

[Beware shorthands silently overriding other styles 8](#_Toc143076902)

[Understanding the order of shorthand values 8](#_Toc143076903)

[Summary 8](#_Toc143076904)

[**Working with relative units** 8](#_Toc143076905)

[**The power of relative values** 8](#_Toc143076906)

[The struggle for pixel-perfect design 8](#_Toc143076907)

[**Ems and rems** 9](#_Toc143076908)

[**EMS FOR FONT SIZE TOGETHER WITH EMS FOR OTHER PROPERTIES** 9](#_Toc143076909)

[**THE SHRINKING FONT PROBLEM** 9](#_Toc143076910)

[**Using rems for font-size** 9](#_Toc143076911)

[**Stop thinking in pixels** 10](#_Toc143076912)

[**Setting a sane default font size** 10](#_Toc143076913)

[**Making the panel responsive** 10](#_Toc143076914)

[**Resizing a single component** 10](#_Toc143076915)

[Viewport-relative units 10](#_Toc143076916)

[Using calc() for font size 10](#_Toc143076917)

[**Unitless numbers and line-height** 11](#_Toc143076918)

[**Custom properties (aka CSS variables)** 11](#_Toc143076919)

[**Changing custom properties dynamically** 11](#_Toc143076920)

[**Changing custom properties with JavaScript** 11](#_Toc143076921)

[Mastering the box model 12](#_Toc143076922)

[**Difficulties with element width** 12](#_Toc143076923)

[**Avoiding magic numbers** 12](#_Toc143076924)

[**Adjusting the box model** 12](#_Toc143076925)

[**Using universal border-box sizing** 12](#_Toc143076926)

[you'll surely run into other elements with the same problem. It would be nice to fix it once, universally 12](#_Toc143076927)

[Adding a gutter between columns 13](#_Toc143076928)

[**Difficulties with element height** 13](#_Toc143076929)

[**Applying alternatives to percentage-based heights** 13](#_Toc143076930)

[**Using min-height and max-height** 14](#_Toc143076931)

[**Negative Margin** 14](#_Toc143076932)

[**Collapsed margins** 14](#_Toc143076933)

[**Collapsing between text** 14](#_Toc143076934)

[**Collapsing outside a container** 15](#_Toc143076935)

[**Spacing elements within a container** 15](#_Toc143076936)

[**Considering changing content** 15](#_Toc143076937)

[Summary 15](#_Toc143076938)

[**Making sense of floats** 16](#_Toc143076939)

[**The purpose of floats** 16](#_Toc143076940)

[**Container collapsing and the clearfix** 16](#_Toc143076941)

[**Understanding container collapsing** 16](#_Toc143076942)

[**Understanding the clearfix** 16](#_Toc143076943)

[**Unexpected “float catching”** 17](#_Toc143076944)

[**Media objects and block formatting contexts** 17](#_Toc143076945)

[**Flexbox** 17](#_Toc143076946)

[**Flexbox principles** 17](#_Toc143076947)

[Adding padding and spacing 17](#_Toc143076948)

[**Flex item sizes** 18](#_Toc143076949)

[**Using the flex-basis property** 18](#_Toc143076950)

[**Using flex-grow** 18](#_Toc143076951)

[**Using flex-shrink** 18](#_Toc143076952)

[**Some Use Cases** 19](#_Toc143076953)

[**Changing the flex direction** 19](#_Toc143076954)

[**Alignment, spacing, and other details** 19](#_Toc143076955)

[**Understanding flex container properties** 19](#_Toc143076956)

[**Understanding flex item properties** 20](#_Toc143076957)

[**Positioning and stacking contexts** 20](#_Toc143076958)

[**Fixed positioning** 21](#_Toc143076959)

[**Absolute positioning** 21](#_Toc143076960)

[**Positioning a pseudo-element** 21](#_Toc143076961)

[**Relative positioning** 21](#_Toc143076962)

[**Stacking contexts and z-index** 21](#_Toc143076963)

[**Understanding the rendering process and stacking order** 22](#_Toc143076964)

[**Manipulating stacking order with z-index** 22](#_Toc143076965)

[**Understanding stacking contexts** 22](#_Toc143076966)

[**Summary** 22](#_Toc143076967)

[Responsive design 23](#_Toc143076968)

[**Mobile first** 23](#_Toc143076969)

[**Adding the viewport meta tag** 23](#_Toc143076970)

[**Media queries** 23](#_Toc143076971)

[**Understanding types of media query** 24](#_Toc143076972)

[**Adding breakpoints to the page** 24](#_Toc143076973)

[BREAKPOINT SELECTION 24](#_Toc143076974)

[**Fluid layouts** 25](#_Toc143076975)

[**Responsive images** 25](#_Toc143076976)

[**Using multiple images for different viewport sizes** 25](#_Toc143076977)

[**Using srcset to serve the correct image** 25](#_Toc143076978)

# Cascade

Predicting how rules behave requires an understanding of the cascade; When two or more rules target the same element on your page, the rules may provide conflicting declarations.

The ***cascade***is the name for this set of rules. It determines how conflicts are resolved, and it’s a fundamental part of how the language works.

* Ex: All three rulesets attempt to set a different font family to this heading. Which one will win? To determine the answer, the browser follows a set of rules, so the result is predictable. In this case, the rules dictate that the second declaration, which has an ID selector, wins;

When declarations conflict, the cascade considers three things to resolve the difference:

* *Stylesheet origin*—Where the styles come from. Your styles are applied in conjunction with the browser’s default styles.
* *Selector specificity*—Which selectors take precedence over which.
* *Source order*—Order in which styles are declared in the stylesheet.

|  |
| --- |
|  |

## Understanding stylesheet origin

The stylesheets you add to your web page aren’t the only ones the browser applies. There are different types, or origins, of stylesheets. Yours are called *author* styles; there are also user agent styles, which are the browser’s default styles. User agent styles have lower priority, so your styles override them.

### USER AGENT STYLES

* Ex: Let’s look again at the example page (figure 1.4). The title is sans-serif because of thestyles you added. A number of other things are determined by the user agent styles: the list has a left padding and a list-style-type of disc to produce the bullets. Linksare blue and underlined.

NOTE You may notice I used ID selectors in this code. There are reasons to avoid doing this;

### IMPORTANT DECLARATIONS

There’s an exception to the style origin rules: declarations that are marked as ***important***. A declaration can be marked important by adding !important to the end of the declaration, before the semicolon:

color: red !important;

## Understanding specificity

If conflicting declarations can’t be resolved based on their origin, the browser next tries to resolve them by looking at their ***specificity***. Understanding specificity is essential. But if you don’t understand specificity, it will bite you. The browser evaluates specificity in two parts: styles applied inline in the HTML and styles applied using a selector.

### INLINE STYLES

If you use an HTML style attribute to apply styles, the declarations are applied only to that element. These are, in effect, “scoped” declarations, which override any declarations applied from your stylesheet or a <style> tag

To override inline declarations in your stylesheet, you’ll need to add an !important to the declaration, shifting it into a higher-priority origin.

### SELECTOR SPECIFICITY

The second part of specificity is determined by the selectors. Different types of selectors also have different specificities. An ID selector has a higher specificity than a class selector, Similarly, a class selector has a higher specificity than a tag selector; The exact rules of specificity are:

* If a selector has more IDs, it wins (that is, it’s more specific).
* If that results in a tie, the selector with the most classes wins.
* If that results in a tie, the selector with the most tag names wins.

NOTE Pseudo-class selectors (for example, :hover) and attribute selectors (for example, [type="input"]) each have the same specificity as a class selector.

The universal selector (\*) and combinators (>, +, ~) have no effect on specificity.

### A NOTATION FOR SPECIFICITY

A common way to indicate specificity is in a number form, often with commas between each number. For example, “1,2,2” indicates a specificity of one ID, two classes, and two tags. IDs having the highest priority are listed first, followed by classes, then tags.

The selector #page-header #page-title has two IDs, no classes, and no tags. We can say this has a specificity of 2,0,0. A specificity of 1,0,0 takes precedence over a specificity of 0,2,2 and even over 0,10,0 because the first number (IDs) is of the higher priority.

### SPECIFICITY CONSIDERATIONS

When you tried to apply the orange background using the .featured selector, it didn’t work. The selector #main-nav a has an ID that overrides the class selector (specificities 1,0,1 and 0,1,0). To correct this, you have some options to consider. Let’s look at several possible fixes. The quickest fix is to add an !important to the declaration you want to favor.

If you start adding !important to multiple declarations, what happens when you need to trump something already set to important?

What if you raised the specificity of your selector? Update the rulesets in your CSS to match this listing.

It is generally best to keep specificity low when you can, so when you need to override something, your options are open.

## Understanding source order

The third and final step to resolving the cascade is source order. If the origin and the specificity are the same, then the declaration that appears later in the stylesheet—or appears in a stylesheet included later on the page—takes precedence.

Although a featured button inside the nav looks correct, what happens if you want to use the

featured class on another link elsewhere on the page, outside of your nav;

When facing a styling problem, I often tackle it in two phases: First figure out what declarations will get it looking right. Second, think through the possible ways to structure the selectors and choose the one that best fits your needs.

### LINK STYLES AND SOURCE ORDER

you may have learned that your selectors for styling links should go in a certain order. If the user hovers over a visited link, the hover styles take precedence. If the user activates the link (that is, clicks it) while hovering over it, the active styles take precedence. A helpful mnemonic to remember this order is LoVe/HAte—link, visited, hover, active.

### CASCADED VALUES

The browser follows these three steps—origin, specificity, and source order to resolve every property for every element on the page. A declaration that “wins” the cascade is called a *cascaded value*.

### Two rules of thumb

there are two common rules of thumb for working with the cascade.

*Don’t use IDs in your selector.* Even one ID ratchets up the specificity a lot. When you need to override the selector, you often don’t have another meaningful ID you can use, so you wind up having to copy the original selector and add another class to distinguish it from the one you are trying to override.

*Don’t use !important.* This is even more difficult to override than an ID, and once you use it, you’ll need to add it every time you want to override the original declaration— and then you still have to deal with the specificity.

# Inheritance

There’s one last way that an element can receive styles—*inheritance*. The cascade is frequently

conflated with the concept of inheritance. Although the two topics are related, you should understand each individually. If an element has no cascaded value for a given property, it may inherit one from

an ancestor element.

# Special values

There are two special values that you can apply to any property to help manipulate the cascade: inherit and initial.

## Using the inherit keyword

Sometimes, you’ll want inheritance to take place when a cascaded value is preventing it. To do this, you can use the keyword inherit. You can override another value with this, and it will cause the element to inherit that value from its parent. You can also use the inherit keyword to force inheritance of a property not normally inherited, such as border or padding.

## Using the initial keyword

Sometimes you’ll find you have styles applied to an element that you want to undo. You can do this by specifying the keyword initial. Every CSS property has an initial, or default, value. If you assign the value initial to that property, then it effectively resets to its default value.

## Shorthand properties

*Shorthand properties* are properties that let you set the values of several other properties at one time. For example, font is a shorthand property that lets you set several font properties like background, border;

### Beware shorthands silently overriding other styles

This can silently override styles you specify elsewhere. If, for example, you were to use the shorthand font property for the page title without specifying a font-weight, a font weight of normal would still be set;

### Understanding the order of shorthand values

**TOP, RIGHT, BOTTOM, LEFT**

Remembering this order can keep you out of trouble. In fact, the word ***TRouBLe***is an mnemonic you can use to remember the order: top, right, bottom, left. Specify three values, and the left and right side will both use the second one. Specify two values, and the top and bottom will use the first one. If you specify only one value, it will apply to all four sides.

**HORIZONTAL, VERTICAL**

Whereas **padding: 1em 2em** specifies the vertical top/bottom values first, followed by the horizontal right/left values, **background-position: 25% 75%** specifies the horizontal right/left values first, followed by the vertical top/bottom values;

## Summary

* Keep selector specificity under control;
* Don’t confuse cascade with inheritance;
* Certain properties are inherited, including those for text, lists, and table borders.
* Don’t confuse initial and auto values;
* Stay out of TRouBLe with shorthand properties;

# **Working with relative units**

One of the most familiar, and probably easiest to work with, is pixels. These are known as ***absolute* units;** that is, 5 px always means the same thing; Other units, such as **em and rem**, are not absolute, but *relative*. The value of relative unit’s changes, based on external factors; Naturally, this makes relative units more difficult to work with.

## **The power of relative values**

### The struggle for pixel-perfect design

In the web environment, the user can have their browser window set to any number of sizes, and the CSS has to apply to it. Furthermore, users can resize the page after it’s opened, and the CSS needs to adjust to new constraints. This means that styles can’t be applied when you create your page; the browser must calculate those when the page is rendered onscreen. This adds a layer of abstraction to CSS. We can’t style an element according to an ideal context;

## **Ems and rems**

Ems are defined by the current element’s font size. the most common relative length unit, are a measure used in typography, referring to a specified font size. In CSS, 1 em means the font size of the current element; its exact value varies depending on the element you’re applying it to;

|  |
| --- |
| .padded {  font-size: 16px;  padding: 1em;  } |

The padding has a specified value of 1em. This is multiplied by the font size, producing a rendered padding of 16 px. This is important: Values declared using relative units are evaluated by the browser to an absolute value, called the *computed value*. In this example, editing the padding to **2 em** would produce a computed value of 32 px.

Using **ems** can be convenient when setting properties like padding, height, width, or border-radius because these will scale evenly with the element if it inherits different font sizes, or if the user changes the font settings.

It’s helpful to know that, for most browsers, the default font size is 16 px.

If you know the **pixel-based font size** you’d like, but want to specify the declaration in ems, here’s a simple formula: divide the desired pixel size by the parent (inherited) pixel size. For example, if you want a 10 px font and your element is inheriting a 12 px font, 10 / 12 = 0.8333 em.

### **EMS FOR FONT SIZE TOGETHER WITH EMS FOR OTHER PROPERTIES**

You’ve now defined ems for font-size (based on an inherited font size). And, you’ve defined ems for other properties like padding and border-radius (based on the current element’s font size). What makes ems tricky is when you use them for both font size and any other properties on the same element. When you do this, the browser must calculate the font size first, and then it uses that value to calculate the other values. Both properties can have the same declared value, but they’ll have different computed values.

What’s happening here is the paragraph inherits a font size of 16 px from the body, producing a calculated font size of 19.2 px. This means that 19.2 px is now the local value for an em, and that value is used to calculate the padding;

### **THE SHRINKING FONT PROBLEM**

Ems can produce unexpected results when you use them to specify the font sizes of multiple nested elements. To know the exact value for each element, you’ll need to know its **inherited font size,** which, if defined on the parent element in ems, requires you to know the parent element’s inherited size, and so on up the tree.

Shrinking text occurs when you **nest** **lists** **several** levels deep and apply an **em-based font size** to each level.

They’re nice for padding, margins, and element sizing, but when it comes to font size, they can get complicated. Thankfully, there is a better option—rems.

## **Using rems for font-size**

When the browser parses an HTML document, it creates Document Object Model. It’s a tree structure, where each element is represented by a node. The <html> element is the top-level (or root) node. Beneath it are its child nodes, <head> and <body>. And beneath those are their children, then their children, and so on. The root node is the ancestor of all other elements in the document. It has a special pseudo-class selector (:root) that you can use to target it;

*Rem* is short for root em. Instead of being relative to the current element, rems are relative to the root element. No matter where you apply it in the document, 1.2 rem has the same computed value: 1.2 times the font size of the root element.

An important part of mastering CSS is learning when to use which tool. My default is to use rems for font sizes, pixels for borders, and ems for most other measures, especially paddings, margins, and border radius;

## **Stop thinking in pixels**

One pattern, or rather, antipattern, that has been common for the past several years is to reset the font size at the page’s root to .625 em or 62.5%. I don’t recommend this. This takes the browser’s default font size, 16 px, and scales it down to 10 px;

There are two problems with this approach. First, it forces you to write a lot of duplicate styles. Ten pixels is too small for most text, so you’ll have to override it throughout the page. The second problem is that when you do this, you’re still thinking in pixels. You might type 1.4 rem into your code, but in your mind, you’re still thinking “14 pixels.”;

### **Setting a sane default font size**

Let’s say you want your default font size to be 14 px. Instead of setting a 10 px default then overriding it throughout the page, set that value at the root. The desired value divided by the inherited value—in this case, the browser’s default—is 14/16, which equals 0.875;

### **Making the panel responsive**

You can use some *media queries* to change the base font size, depending on the screen size. This’ll make the panel render at different sizes based on the size of the user’s screen;

### **Resizing a single component**

You can also use ems to scale an individual component on the page. Sometimes you might need a larger version of the same part of your interface on certain parts of the page. All you have to do is override the parent element’s 1 rem with another value. Because all the component’s measurements are relative to this, overriding it will resize the entire panel.

## Viewport-relative units

You’ve learned that ems and rems are defined relative to font-size, but these aren’t the only type of relative units. There are also *viewport-relative units* for defining lengths relative to the browser’s viewport

50 vw is equal to half the width of the viewport, and 25 vh equals 25% of the viewport’s height. vmin is based on which of the two (height or width) is smaller. This is helpful for ensuring that an element will fit on the screen regardless of its orientation: If the screen is landscape, it’ll be based on the height; if portrait, it’s based on the width.

The viewport-relative lengths are great for things like making a large hero image fill the screen. Your image can be inside a long container, but setting the image height to 100 vh, makes it exactly the height of the viewport.

### Using calc() for font size

The calc() function lets you do basic arithmetic with two or more values. This is particularly useful for combining values that are measured in different units. This function supports addition (+), subtraction (-), multiplication (\*) and division (/). The addition and subtraction operators must be surrounded by whitespace;

:root {

font-size: calc(0.5em + 1vw);

}

## **Unitless numbers and line-height**

Some properties allow for *unitless* values (that is, a number with no specified unit). Properties that support this include line-height, z-index, and font-weight; You can also use the unitless

value 0 anywhere a **length** unit. You’ll specify a line height for the body element and allow it to be inherited by the rest of the document. This will work as expected, no matter what you do to the font sizes

in the page;

|  |
| --- |
| body {  line-height: 1.2;  }  .about-us {  font-size: 2em;  } |

The paragraph inherits a line height of 1.2. Because the font size is 32 px (2 em × 16 px, the browser’s default), the line height is calculated locally to 38.4 px (32 px × 1.2). This will leave an appropriate

amount of space between lines of text.

If instead you specify the line height using a unit, you may encounter unexpected results,

## **Custom properties (aka CSS variables)**

This specification introduced the concept of variables to the language, which enabled a new level of dynamic, context-based styles. You can declare a variable and assign it a value; then you can

reference this value throughout your stylesheet.

:root {

--main-font: Helvetica, Arial, sans-serif;

}

This listing defines a variable named --main-font, and sets its value to a set of common

sans-serif fonts. The name must begin with two hyphens (--) to distinguish it from CSS properties, followed by whatever name you’d like to use. Variables must be declared inside a declaration block. I’ve used the :root selector here, which sets the variable for the whole page; function called var() allows the use of variables. You’ll use this function to reference the --main-font variable just defined;

The var() function accepts a second parameter, which specifies a fallback value. If the variable specified in the first parameter is not defined, then the second value is used instead.

### **Changing custom properties dynamically**

But what makes them particularly interesting is that the declarations of custom properties cascade and inherit: You can define the same variable inside multiple selectors, and the variable will have a different value for various parts of the page; You can define a variable as black, for example, and then redefine it as white inside a particular container. Then, any styles based on that variable will dynamically resolve

to black if they are outside the container and to white if inside. The custom properties behave as a sort of scoped variable because the values are inherited by descendant elements. Inside the dark container, --main-color is white; elsewhere on the page, it’s black.

## **Changing custom properties with JavaScript**

Custom properties can also be accessed and manipulated live in the browser using JavaScript.

|  |
| --- |
| <script type="text/javascript">  var rootElement = document.documentElement;  var styles = getComputedStyle(rootElement);  var mainColor = styles.getPropertyValue('--main-bg');  console.log(String(mainColor).trim());  </script>  “””  var rootElement = document.documentElement;  rootElement.style.setProperty('--main-bg', '#cdf');  “”” |

Summary

* Embrace the use of relative units, allowing the page’s structure to determine the meaning of your styles.
* Favor the use of rems for font size, but selectively use ems for simple scaling of components on the page.
* You can make your entire page scale responsively without any media queries.
* Use unitless values when specifying line height.
* You can start getting familiar with one of CSS’s newest features, custom properties. Be aware that any declaration using var() will be ignored by old browsers that don’t understand it. Provide a fallback behavior for those browsers when possible: color: black; color: var(--main-color);

# Mastering the box model

it’s important to have a solid grasp on the fundamentals of how the browser sizes and positions elements. We’ll look at some of the edge cases of the box model, and I’ll give you practical advice for sizing and aligning elements. We’ll also tackle two of the most notorious problems in CSS: vertical centering and equal-height columns.

## **Difficulties with element width**

NOTE IE has a bug where <main> elements are rendered inline by default, rather than as blocks. We corrected that here by adding a **display: block** declaration.

let’s put your two columns in place. To begin, you’ll use a float-based layout. You’ll float the main and the sidebar to the left and give them widths of 70% and 30%, respectively.

Instead of the two columns sitting side by side, they line wrapped. Even though you specified widths of 70% and 30%, the columns took up more than 100% of the available space. That’s because of the default behavior of the box model; When you set the width or height of an element, you’re specifying the width or height of its content; any padding, border, and margins are then added to that width. This behavior means that an element with a 300 px width, a 10 px padding, and a 1 px border has a rendered width of 322 px;

### **Avoiding magic numbers**

The naive fix is to reduce the width of one of the columns (the sidebar, for example). On my screen, a width of 26% for the sidebar works, but this is unreliable. The 26% is known as a *magic number.* Instead of using a desired value, I found it by making haphazard changes to my styles until I got the result I wanted. For programming in general, magic numbers aren’t desirable;

### **Adjusting the box model**

CSS allows you to adjust the box model behavior with its **boxsizing** property. By default, box-sizing is set to the value of content-box. This means that any height or width you specify only sets the size of the content box. You can assign a value of border-box to the box sizing instead. That way, the height and width properties set the combined size of the content, padding, and border, which is exactly what you want in this example.

### **Using universal border-box sizing**

### you'll surely run into other elements with the same problem. It would be nice to fix it once, universally

for all elements, You can do this with the universal selector (\*), which targets all elements on the page;

\*,

::before,

::after {

box-sizing: border-box;

}

|  |
| --- |
| :root {  box-sizing: border-box;  }  \*,  ::before,  ::after {  box-sizing: inherit;  } |

## Adding a gutter between columns

It’s often more visually appealing to have a small gap (or *gutter*) between columns. prefer an em-based gutter, which I find more consistent. You can accomplish this with calc(). you can move 1.5 em.

This listing shows how calc() makes this possible. Change your CSS again to match this listing.

**width: calc(30% - 1.5em);**

## **Difficulties with element height**

Typically, it’s best to avoid setting explicit **heights** on elements. Normal document flow is designed to work with a **constrained width and an unlimited height**. Contents fill the width of the viewport and then line wrap as necessary. Because of this, the height of a container is organically determined by its contents, not by the container itself.

**Controlling overflow behavior**

When you explicitly set an element’s height, you run the risk of its contents *overflowing* the container. This happens when the content doesn’t fit the specified constraint andrenders outside of the parent element; You can control the exact behavior of the overflowing content with the overflow property,

**visible**, **hidden**, **scroll**, and **auto;**

Browsers insert a scrollbar for scrolling the page, and adding nested scrollable areas inside your page can be frustrating to users.

### **Applying alternatives to percentage-based heights**

Specifying height using a percentage is problematic. Percentage refers to the size ofan element’s containing block; the height of that container, however, is typically determinedby the height of its children. This produces a circular definition that the browser can’t resolve, so it’ll ignore the declaration. For percentage-based heights to work, the parent must have an explicitly defined height. A better approach is to use the viewport-relative vh units,

#### COLUMNS OF EQUAL HEIGHT

The columns-of-equal-height problem is one weakness that has plagued CSS from the beginning. if you allowed the columns to determine their heights naturally, each element would evaluate to a different height, based on its content;

Modern browsers make it much easier— they support CSS tables. supports **display: table,** and IE10+ allows for a flexible box, or **flexbox**, both of which, by default, produce equal-height columns;

#### CSS TABLE LAYOUTS

First, you’ll use a CSS-based table layout. Instead of using floats, you’ll make the container a display: table and each column a display: table-cell;

To define space between cells of a table, you can use the border-spacing property of the table element. This property accepts two length values: one for horizontal spacing and one for vertical spacing.

You could add border-spacing: 1.5em 0 to your container, but this has a peculiar side effect: that value is also applied to the outside edges of the table.

You can fix this with the clever use of a *negative* margin, but that needs to go on a new container that wraps around the whole table. Here’s how. Add a <div class="wrapper"> around the container and apply a left and right margin of -1.5 em to counteract the 1.5 em of the border spacing on the sidebars.

**WARNING Never explicitly set the height of an element unless you have noother choice. Always seek an alternative approach first. Setting a height invariably leads to further complications.**

### **Using min-height and max-height**

Two properties that can be immensely helpful are min-height and max-height. Instead of explicitly defining a height, you can use these properties to specify a minimum or maximum value, allowing the element to size naturally within those bounds.

Suppose you want to place your hero image behind a larger paragraph of text, you can specify a minimum height with min-height. This means the element will be at least as high as you specify, and if the content doesn't fit, the browser will allow the element to grow naturally to prevent overflow. max-height allows an element to size naturally, up to a point. If that size is reached, the element doesn’t become any taller, and the contents will overflow.

**Vertically centering content**

Vertical centering in CSS is another notorious problem. Why doesn’t vertical-align work? Developers are often frustrated when they apply vertical-align: middle to a block element, expecting it to center the contents of the block. A **vertical-align** declaration only affects inline and table-cell elements. With inline elements, it controls alignment among other elements on the same line.

the simplest way to vertically center in CSS—give a container equal top and bottom padding, and let both the container and its contents determine their height naturally; This approach works whether the content inside the container is inline, block, or of any other display value.

The best approach to centering contents inside a container may depend on a number of factors based on your particular scenario.

* Can you use a natural height container? Apply an equal top and bottom padding to the container to center its contents.
* Do you need a specific height container, or do you need to avoid using padding? Use **display: table-cell and vertical-align: middle** on your container.
* you can center your content with flexbox. See chapter 5.

## **Negative Margin**

This has some peculiar uses, such as allowing elements to overlap or stretch wider than their containers. The exact behavior of a negative margin depends on which side of the element you apply it to. Negative left or top margins pull the element leftward or upward. Adding negative right or bottom margins will pull the succeeding element(s) leftward or upward (to overlap);

**WARNING Using negative margins to overlap elements can render some elements unclickable if they’re moved beneath other elements.**

## **Collapsed margins**

When top and/or bottom margins are adjoining, they overlap, combining to form a single margin. This is referred to as *collapsing*.

### **Collapsing between text**

The main reason for collapsed margins has to do with the spacing of blocks of text. Paragraphs (<p>), by default, have a 1 em top margin and a 1 em bottom margin. This is applied by the user agent stylesheet. But when you stack two paragraphs, one after the other, their margins don’t add up to a gap of 2 em. Instead they collapse, overlapping to produce only 1 em of space between the two paragraphs. The size of the collapsed margin is equal to the largest of the joined margins.

NOTE Margin collapsing only occurs with top and bottom margins. Left and right margins don’t collapse.

**Collapsing multiple margins**

Elements don’t have to be adjacent siblings for their margins to collapse. Even if you wrap the paragraph inside an extra div, as in the next listing, the visual **result will be the same**. In this case, there are three different margins collapsing together: the bottom margin of the <h2>, the top margin of the <div>, and the top margin of the <p>. so the space between the elements is still 19.92 px, the **largest** of the three.

### **Collapsing outside a container**

The way three consecutive margins collapse might catch you off guard. An element's margin collapsing outside its container typically produces an undesirable effect if the container has a background.

Here are ways to prevent margins from collapsing:

* Applying overflow: auto (or any value other than visible) to the container prevents margins inside the container from collapsing with those outside the container.
* Adding a border or padding between two margins stops them from collapsing.
* Margins won’t collapse to the outside of a container that is floated, that is an inline block, or that has an absolute or fixed position.
* When using a **flexbox**, margins won’t collapse between elements that are part ofthe flex layout.

## **Spacing elements within a container**

No matter which approach you choose, however, you’ll still encounter a problem: the margin needs to work in conjunction with the sidebar’s padding. If you add **margin-top: 1.5em,**

You can fix this in a number of ways. Listing 3.17 shows one of the simpler fixes. It uses the adjacent sibling combinator (+) to target only button-links that immediately follow other button-links as siblings under the same parent element. Now the margin only appears between two buttons.

.button-link + .button-link {

margin-top: 1.5em;

}

Only apply a top margin to button-links that immediately follow another button-link;

### **Considering changing content**

You’re on the right track, but the spacing problem arises again as soon as you add more content to the sidebar. Spacing is off between the second button and the bottom link.

be.” Instead of fixing margins for the current page contents, let’s fix it in a way that works no matter how the page gets restructured. You’ll do this with something Pickering calls a *lobotomized owl selector*. It

looks like this: \* + \*.

That’s a universal selector (\*) that targets all elements, followed by an adjacent sibling combinator (+), followed by another universal selector. Except, instead of targeting buttons that immediately follow other buttons, it targets any element that immediately follows any other element. That is, it selects all elements on the page that aren’t the first child of their parent;

body \* + \* {

margin-top: 1.5em;

}

Using the lobotomized owl like this is a tradeoff. It simplifies many margins throughout your page, but you’ll have to override it in places where you don’t want it to apply. The lobotomized owl may not be the correct solution for every project, and it’s difficult to add to an existing project without breaking the layout,

## Summary

* Always use a universal border-box fix for predictable element sizing.
* Avoid explicitly setting the height of an element to avoid overflow issues.
* Use modern layout techniques like display: table or a flexbox to produce columns of equal height or to vertically center content.
* If your margins behave oddly, take steps to prevent margins from collapsing.
* Consider using the lobotomized owl selector on your page to globally apply margins between stacked elements.

# **Making sense of floats**

## **The purpose of floats**

Although floats were not originally intended to construct page layouts. A ***float***pulls an element (often an image) to one side of its container, allowing the document flow to wrap around it. We don’t always use floats in this way, even though it’s their original purpose.

**TIP It’s usually easiest to lay out the large regions of a page first, then work your way to the smaller elements within.**

By using max-width instead of width, the element shrinks to below 1080 px if thescreen’s viewport is smaller than that. That is to say, in smaller viewports, the inner container will fill the screen,

And floats are still the only way to move an image to the side of the page and allow text to wrap around it.

## **Container collapsing and the clearfix**

### **Understanding container collapsing**

On your page, let’s float the four media boxes to the left. The problems will immediately become apparent; What happened to the white background? You’ve set a light gray background on each media box, expecting to see the white background of the container behind (or rather, around) them. Instead, the white background stopped above the top row of media boxes. Why is this?

The float in one container extends into the next container, allowing text in both containers to wrap around the floated element;

In your page, everything inside the main element is floated except for the page title, so only the page title contributes height to the container, leaving all the floated media elements extending below the white background of the main. This isn’t the behavior we want, so let’s fix it.

### **Understanding the clearfix**

Instead of adding an extra div to your markup, you’ll use a *pseudo-element*. By using the **::after** pseudo-element selector, you can effectively insert an element into the DOM at the end of the container, without adding it to the markup.

Note pseudo-elements are ::before and ::after, which are used to insert content at the beginning or end of an element;

It’s important to know that the clearfix is applied to the element that contains the floats; the space in the

content value. An empty string ("") works as well, but the added space character fixes an obscure bug in older versions of Opera.

|  |
| --- |
| .clearfix::before,  .clearfix::after {  display: table;  content: " ";  }  .clearfix::after {  clear: both;  } |

TIP This version of the clearfix also doubles as a useful way to prevent margin collapsing where we don’t want it.

## **Unexpected “float catching”**

Three left-floated boxes: Box 3 doesn’t float all the way to the left if box 1 is taller than box 2, instead it floats up against box 1.

Because box 2 is shorter than box 1, there’s room for box 3 beneath it. Instead of clearing box 1, box 3 “catches” on it. The exact nature of this behavior is dependent on the heights of each of the floated blocks.

The fix for this is simple: The third float needs to clear the floats above it. Or, more generally, the first element of each row needs to clear the float above it. Because you know you have two boxes per row, You can target these with the :nth-child() pseudo-class selector;

NOTE This technique for clearing each row only works when you know how many elements are on each row.

## **Media objects and block formatting contexts**

The media object pattern: an image on the left and descriptive content on the right;

# **Flexbox**

## **Flexbox principles**

Flexbox begins with the familiar display property. Applying display: flex to an element turns it into a *flex container*, and its direct children turn into *flex items*. The flex container fills the available width like a block element; The flex items are all the **same** height, determined naturally by their contents. a flex container asserts control over the layout of the elements within;

I’ve used direct descendant combinators (>) to ensure you only target direct child elements;

### Adding padding and spacing

you’ll apply the menu item padding to the internal <a> elements, not the <li> elements. You’ll need the entire area that looks like a menu link to behave like a link when the user clicks it.

You’ll notice you made the links a display block. If they were to remain inline, the height they’d contribute to their parent would be derived from their line height—not their padding and content, which is the behavior you want for this page. You also applied a little more horizontal padding than vertical, which is generally more aesthetically pleasing.

flexbox allows you to use margin: auto to fill available spacebetween the flex items. You can also use this to move the final menu item to the right side; where you’ll apply a margin between each item, but not to the outside edges. You can achieve this layout by using the margin-left property and an adjacent sibling combinator; You’ll also apply an auto left margin to the last button, which causes the margin to fill all the available space, pushing the last button all the way to the right

|  |  |
| --- | --- |
| .site-nav > li + li {  margin-left: 1.5em;  }  .site-nav > .nav-right {  margin-left: auto;  } | **Targets every list item that follows another list item**  **(that is, all but the first)**  **Auto margins inside a flexbox will fill the available space.** |

## **Flex item sizes**

The **flex** property controls the size of the flex items along the main axis. You haven’t done anything yet to specify the width of the two columns, so they’ll size themselves naturally, based on their content.

NOTE When it comes to CSS, it’s important to consider not only the specific content you have on the page now, but also what will happen as that content changes;

The flex property, which is applied to the flex items, gives you a number of options. You’ll use the column-main and column-sidebar classes to target the columns, using flex to apply widths of two thirds and one-third. with the main column twice as wide as the sidebar;

The flex property is shorthand for three different sizing properties: flex-grow, flex-shrink, and flex-basis. In this listing, you’ve only supplied flex-grow, leaving the other two properties to their default values (1 and 0% respectively). **So flex: 2 is equivalent to flex: 2 1 0%.**

We’ll start with flexbasis, as the other two are based on it;

### **Using the flex-basis property**

The *flex basis* defines a sort of starting point for the size of an element—an initial “main size.” The flex-basis property can be set to any value that would apply to width, including values in px, ems, or percentages. Once this initial main size is established for each flex item, they may need to grow or

shrink in order to fit (or fill) the flex container along the main axis.

### **Using flex-grow**

The remaining space (or remainder) will be consumed by the flex items based on their flex-grow values, which is always specified as a non-negative integer. If an item has a flex-grow of 0, it won’t grow past its flex basis. If any items have a non-zero growth factor, those items will grow until all of the remaining space is used up;

higher flex-grow value gives that element more “weight”; it’ll take a largerportion of the remainder. An item with flex-grow: 2 will grow twice as much as an item with flex-grow: 1;

flex: 2 and flex: 1 set a flex basis of 0%, so 100% of the container’s width is the remainder (minus the 1.5 em margin between the two columns). The remainder is then distributed to the two columns: two-thirds to the first column and the remaining third to the second;

**TIP Favor the use of the shorthand flex property instead of individually declaring flex-grow, flex-shrink, or flex-basis. The shorthand assigns useful default values for any of the three that you omit: flex-grow of 1, flex-shrink of 1, and a flex-basis of 0%. These are mostcommonly what you’ll need.**

### **Using flex-shrink**

After determining the initial main size of the flex items, **they could exceed the size available in the flex container. Without flex-shrink, this would result in an overflow; --** Flex items can have an initial size exceeding that of the flex container;

The flex-shrink value for each item indicates whether it should shrink to prevent overflow. If an item has a value of flex-shrink: 0, it will not shrink. Items with a value greater than 0 will shrink until there is no overflow. An item with a higher value will shrink more than an item with a lower value, proportional to the flex-shrink values.

### **Some Use Cases**

You can make use of the flex property in countless ways. You can define proportional columns using flex-grow values or flex-basis percentages as you did on your page. You can define fixed width columns and “fluid” columns that scale with the viewport. The third example illustrates part of the “Holy Grail” layout. This is a layout that has been notoriously difficult in CSS. The two sidebars are a fixed width, whereas the center column is “fluid,”;

Flex direction

Another important option in flexbox is the ability to shift the direction of the axes. The flex-direction property, applied to the flex container, controls this. Its initial value (row) causes the items to flow left-to-right, as you’ve done. Specifying flexdirection: column causes the flex items to stack vertically (top to bottom) instead. Changing the flex direction changes the main axis. The cross axis changes as well, to remain perpendicular to the main axis.

### **Changing the flex direction**

What you need is for the two columns to grow if necessary to fill the container’s height. To do this, turn the right column (the column-sidebar) into a flex container with a flex-direction: column. Then, apply a non-zero flex-grow value to both tiles within.

The inner flexbox here has a flex direction of column, so the main axis is rotated. Itflows from top to bottom (and the cross axis now flows from left to right). This meansthat for those flex items, flex-basis, flex-grow, and flex-shrink now apply to theelement height rather than the width. By specifying flex: 1, the height of these items will stretch if necessary to fill the container.

I’ve combined the :not() pseudo-class with the attribute selectors [type=checkbox]and [type=radio] (see appendix A for details). This targets all input elements except checkboxes and radio buttons.

## **Alignment, spacing, and other details**

In general, you’ll begin a flexbox with the methods we’ve already covered:

* Identify a container and its items and use display: flex on the container
* If necessary, set the flex-direction on the container
* Declare margins and/or flex values for the flex items where necessary to control their size;s

### **Understanding flex container properties**

* flex-direction This specifies the direction of the main axis. The cross axis will be perpendicular to the main axis.
* flex-wrap This specifies whether flex itemswill wrap on to a new row inside the flex container;
* flex-flow Shorthand for <flex-direction> <flex-wrap>
* justify-content Controls how items are positioned along the main axis;
* align-items Controls how items are positioned along the cross axis.
* align-content If flex-wrap is enabled, this controls the spacing of the flex rows along the cross axis. If items don’t wrap, this property is ignored.
* flex-grow An integer that specifies the “growth factor,” determining how much the item will grow along the main axis to fill unused space;
* flex-shrink An integer that specifies the “shrink factor,” determining how much the item will shrink along the main axis, if needed, to prevent overflow. Ignored if the container has flex wrap enabled.
* flex-basis Specifies the initial size of the item before flex-grow or flex-shrink is applied.
* flex Shorthand for: <flex-grow> <flex-shrink> <flex-basis>
* order An integer that moves a flex item to a specific position among its siblings, disregarding source order.
* align-self Controls how the item is aligned on the cross axis. This will override the container’s align-items value for specific item(s). Ignored if the item has an auto margin set on the cross axis

JUSTIFY-CONTENT PROPERTY

The justify-content property controls how the items are spaced along the main axis if they don’t fill the size of the container.

ALIGN-ITEMS PROPERTY

Whereas justify-content controls item alignment along the main axis, align-items adjusts their alignment along the cross axis. The initial value for this is stretch, which causes all items to fill the container’s height in a row layout, or width in a column layout. This provides columns of equal height.

ALIGN-CONTENT PROPERTY

If you enable wrapping (using flex-wrap), this property controls the spacing of each row inside the flex container along the cross axis. Supported values are flex-start, flex-end, center, stretch (the initial value), space-between, and space-around;

### **Understanding flex item properties**

ALIGN-SELF PROPERTY

This property controls a flex item’s alignment along its container’s cross axis. This does the same thing as the flex container property align-items, except it lets you align individual flex items differently.

ORDER PROPERTY

Normally, flex items are laid out in the order they appear in the HTML source. They are stacked along the main axis, beginning at the start of the axis. By using the order property Initially, all flex items have an order of 0. Specifying a value of -1 to one item will move it to the beginning of the list, and a value of 1 will move it to the end.

Summary

* Use flexbox for versatile, easy-to-control layout of page content.
* Autoprefixer can simplify flexbox support for older browsers.
* Use flex to specify almost any imaginable combination of flex item sizes.
* Use nested flexboxes to piece together more complicated layouts and to fill the heights of naturally sized boxes.
* Flexbox automatically creates columns of equal height.
* Use align-items or align-self to vertically center a flex item inside its flex container.

# **Positioning and stacking contexts**

The initial value of the position property is static. Everything we’ve done in previous chapters is with static positioning. When you change this value to anything else, the element is said to be *positioned*. An element with static positioning is thus *not positioned*. This allows you to place the element somewhere

else on the screen. It can place elements in front of or behind one another, thus overlapping one another.

## **Fixed positioning**

Fixed positioning, applying position: fixed to an element lets you position the element arbitrarily within the viewport. This is done with four companion properties: top, right, bottom, and left. The values you assign to these properties specify how far the fixed element should be from each edge of the browser viewport.

**Controlling the size of positioned elements**

When positioning an element, you’re not required to specify values for all four sides. You can specify only the sides you need and then use width and/or height to help determine its size. You can also allow the element to be sized naturally.

## **Absolute positioning**

Absolute positioning works the same way, except it has a different containing block. Instead of its position being based on the viewport, its position is based on the closest positioned ancestor element. As with a fixed element, the properties top, right, bottom, and left place the edges of the element within its containing block.

NOTE If none of the element’s ancestors are positioned, then the absolutely positioned element will be positioned based on something called the *initial* *containing block*. This is an area with dimensions equal to the viewport size;

### **Positioning a pseudo-element**

Instead, you can use CSS to hide the word *close* and display an *x*. You’ll accomplish this by doing two things. First, you’ll push the button’s text outside the button and hide the overflow. Second, you’ll use the content property to add the *x* to the button’s ::after pseudo-element and absolute positioning to center it within the button.

This listing explicitly sets the button size to 1 em square. The text-indent property then pushes the text to the right, outside of the element. The exact value doesn’t matter as long as it’s more than the width of the button. Then, because text-indent is an inherited property, you reset it to 0 on the pseudo-class so the *x* isn’t also indented. The pseudo-class is now absolutely positioned.

## **Relative positioning**

The top, right, bottom, and left properties, if applied, will shift the element from its original position, but it won’t change the position of any elements around it. I’ve applied three additional properties to the second element: position: relative; top: 1em; left: 2em. As you can see, this has shifted the element from its initial position; These shifts may cause that element to overlap elements below or beside it.

NOTE Unlike fixed and absolute positioning, you cannot use top, right, bottom, and left to change the size of a relatively positioned element. Those values will only shift the position of the element up or down, left or right;

Far more often, you’ll use position: relative to establish the containing block for an absolutely positioned element inside it.

## **Stacking contexts and z-index**

Eventually, you’ll encounter problems with **stacking**. When you position multiple elements on the same page, you may run into a scenario where two different positioned elements overlap. You may occasionally be surprised to find the “wrong” one appearing in front of the other;

### **Understanding the rendering process and stacking order**

It’s also responsible for determining the order in which the browser will *paint* the elements. This order is important because elements painted later appear in frontof elements painted earlier, should they happen to overlap.

Under normal circumstances (that is, before positioning is applied), this order is determined by the order the elements appear in the HTML. The browser first paints all non-positioned elements, then it paints the positioned ones.

One way to fix your page would be to move the <div class="modal"> and all of its contents to somewhere after the dropdown menu.

Relative positioning depends on the document flow, and absolute positioning depends on its positioned ancestor element. We need a way to control their stacking behavior. This is done through a property called z-index.

### **Manipulating stacking order with z-index**

The z-index property can be set to any integer (positive or negative). Elements with a higher z-index appear in front of elements with a lower z-index. Elements with a negative z-index appear behind static elements.

Z-index seems straightforward, but using it introduces two gotchas. First, z-index only works on positioned elements. You cannot manipulate the stacking order of static elements. Second, applying a z-index to a positioned element establishes something called a stacking context;

### **Understanding stacking contexts**

A *stacking context* consists of an element or a group of elements that are painted together by the browser. One element is the root of the stacking context, so when you add a z-index to a positioned element that element becomes the root of a new stacking context.

All of its descendant elements are then part of that stacking context. Stacking contexts deal with which elements are in front of other elements; if an element is stacked in front of a stacking context, no element within that stacking context can be brought in front of it.

This markup consists of three boxes, two of which will be positioned and given a zindex of 1. The absolute element inside the first box will be positioned and given a zindex of 100. Despite its high z-index, it still appears behind the second box because its parent forms a stacking context behind the second box;

If you ever find that z-index isn’t behaving how you’d expect, look up the DOM tree at the element’s ancestors until you find one that is the root of a stacking context.

Note: You should fight this urge. The more you use positioning, the more complicated the page becomes, and the harder it is to debug. Generally speaking, you should only do this when you need to stack elements in front of one another.

**Sticky positioning**

It’s sort of a hybrid between relative and fixed positioning: The element scrolls normally with the page until it reaches a specified point on the screen, at which point it will “lock” in place as the user continues to scroll.

## **Summary**

* Use fixed positioning for modal dialogs.
* Use absolute positioning for dropdown menus, tooltips, and other dynamic interactions
* Be aware of accessibility concerns when building these features.
* There are two gotchas of z-index: it only works on positioned elements and using it creates a new stacking context.
* Be aware of the potential pitfalls when creating multiple stacking contexts on a page.
* Keep an eye out for better browser support of sticky positioning;

# Responsive design

The three key principles to responsive design:

* *A mobile first approach to design*. This means you build the mobile version before you construct the desktop layout.
* *The* @media *at-rule*. With this rule, you can tailor your styles for viewports of different sizes. This syntax (often called *media queries*) lets you write styles that only apply under certain conditions.
* *The use of fluid layouts*. This approach allows containers to scale to different sizes based on the width of the viewport.

## **Mobile first**

The first principle of responsive design is *mobile first*. As mentioned, this means exactly what it sounds like: You build your mobile layout before you build the desktop. This is the best way to ensure both versions work;

**IMPORTANT** When writing the HTML for a responsive design, it’s important to ensure it has everything you need for each screen size. You can apply different CSS for each instance, but they must all share the same HTML;

*Breakpoint* —A particular point at which the page styles change to provide the best possible layout for the screen size.

The text-shadow property in the hero image might be new to you. It consists of several values that together define a shadow to add behind the text. The first two values are Cartesian coordinates, indicating how far the shadow should shift from the text’s position. The values 0.1em 0.1em shift the shadow slightly right and down. The third value (0.3em) indicates how much to blur the shadow.

TIP When designing for mobile touchscreen devices, be sure to make all the key action items large enough to easily tap with a finger. Don’t make your users zoom in in order to tap precisely on a tiny button or link.

### **Adding the viewport meta tag**

the viewport *meta tag*. This is an HTML tag that tells mobile devices you’ve intentionally designed for small screens. Without it, a mobile browser assumes your page is not responsive;

## **Media queries**

*Media queries* allow you to write a set of styles that only apply to the page under certain conditions.This lets you tailor your styles differently, based on the screen size. You can define a setof styles that apply to small devices, another set for medium-sized devices, and yet athird set for large screens to allow for laying out parts of the page differently.Media queries use the @media at-rule to target devices that match a specified feature.A basic media query looks like this:

@media (min-width: 560px) {

.title > h1 {

font-size: 2.25rem;  
 }  
}

The padding will only be applied to a page-header element if the user’s device has a viewport width of 560 px or greater. If the viewport is less than this, the rules inside are ignored. The rules inside a media query still follow the normal rules of the cascade. They can override rules outside of the media query;

Note I used px in the example, but it’s a better idea to use ems in your media queries, based on the browser’s default font size (usually 16 px). Instead of 560 px, you should use 35 em (560 / 16);

### **Understanding types of media query**

You can further refine a media query by joining the two clauses with the keyword “and”

@media (min-width: 20em) and (max-width: 35em) { … } This combined media query only targets devices that meet both criteria;

If you want a media query that targets one of multiple criteria, use a comma: @media (max-width: 20em), (min-width: 35em) { … } This example targets both viewports 20 em and narrower and those 35 em and wider.

you’ve used min-width, which targets devices with a viewport above a certain width, and max-width, which targets devices below a certain width. These are each called a *media feature*.

min-width and max-width are the most common ones you’ll use by far. But you can also use a number of other types of media features.

* (min-height: 20em)—Targets viewports 20 em and taller
* (max-height: 20em)—Targets viewports 20 em and shorter
* ( orientation: landscape)—Targets viewports that are wider than they are tall
* orientation: portrait)—Targets viewports that are taller than they are wide

MEDIA TYPES

The two media types you’ll generally need to think about are screen and print. Using a print media query lets you control how your page will lay out if the user prints the page, so you can do things like removing background images (to save on ink) and hiding unneeded navigation. To write print styles that apply only when printing, use the query @media print. No parentheses are necessary as with min-width and other media features. To target screen only, use @media screen.

### **Adding breakpoints to the page**

Practically speaking, a mobile-first approach means the type of media query you’ll use the most should be min-width. You’ll write your mobile styles first, outside of any media queries.

|  |  |
| --- | --- |
| .title {  ...  } | **Mobile styles; applied to all breakpoints** |
| @media (min-width: 35em) {  .title {  ...  }  } | **Medium breakpoint; overrides select mobile styles** |
| @media (min-width: 50em) {  .title {  ...  }  } | **Large breakpoint; overrides select small and medium**  **breakpoint styles** |

You might be wondering how I arrived at a breakpoint of 35 em in listing 8.7. I chose it because this is the point where the three columns started to feel overcrowded. In this case, below 35 ems, the columns

were too narrow.

### BREAKPOINT SELECTION

Most of the time, you’ll want to start setting breakpoints with the parts of your design that have multiple columns. Try a number of breakpoints until you find one that feels right. Ensure your columns aren’t too narrow above that breakpoint. You’ll find hundreds of devices with hundreds of different screen resolutions; you’ll never test them all. Choose the breakpoints that make sense for your design, and it’ll play outwell, regardless of the device a user has.

### **Fluid layouts**

Fluid layout (sometimes called *liquid layout*) refers to the use of containers that grow and shrink according to the width of the viewport. This is in contrast to a fixed layout, where columns are defined using pixels or ems. A fixed container (for example, one with width: 800px) will overflow the viewport on smaller devices, forcing the need for horizontal scrolling. A fluid container automatically shrinks to fit.

Note: Do make it a habit to think of container widths in percentages rather than in any fixed size.

Dealing with tables

Tables are particularly problematic for fluid layout on mobile devices. If a table has more than a handful of columns, it can easily overflow the screen width; One approach you can take is to force the table to display as normal block elements.

## **Responsive images**

In responsive design, images need special attention. you must also consider the bandwidth limitations of mobile users. Images tend to be among the largest resources used on a page. The first thing you should do is always make sure your images are well compressed. Use the Save for web option in your image editor, which will greatly reduce the image’s file size, or use another image compression tool such as https://tinypng.com/.

You should also ensure they’re not any higher resolution than necessary. Determining what “necessary” means, however, depends on the viewport size. You don’t need to serve as large a file to smaller screens because they’ll be scaled down anyway.

### **Using multiple images for different viewport sizes**

The best practice is to create a few copies of an image, each at a different resolution. If you know, based on media queries, that the screen is a certain size, there’s no sense sending an extremely large image; the browser will have to downscale it to make it fit.

|  |
| --- |
| background-image: url(coffee-beans-small.jpg); // **smallest image on mobile devices** |
| background-image: url(coffee-beans-medium.jpg); // **Uses a larger image on medium-size screens** |
| background-image: url(coffee-beans.jpg); // **Uses the full resolution image on large screens** |

If you load this in your browser, you won’t notice a difference at all. And that’s exactly the point. If you’re on a small breakpoint, your screen isn’t wide enough to show the full resolution image anyway.

Summary

* Always build your designs mobile first.
* Use media queries to progressively enhance the page at larger and larger viewports.
* Use fluid layouts that fit the screen at any browser size.
* Use responsive images to fit the bandwidth limitation of mobile devices.
* Don’t forget to include your meta viewport tag.

### **Using srcset to serve the correct image**

Media queries solve the problem when the image is included via the CSS, but what about images added via the HTML <img> tag? For inlined images, a different approach is necessary: the srcset attribute;

|  |
| --- |
| <img  alt="A white coffee mug on a bed of coffee beans"  src="coffee-beans-small.jpg"  srcset="coffee-beans-small.jpg 560w,coffee-beans-medium.jpg 800w,coffee-beans.jpg 1280w"  /> |

TIP As part of a fluid layout, you should always ensure images don’t overflow their container’s width. Do yourself a favor and always add this rule to your stylesheet to prevent that from happening: img { max-width: 100%; }.

# **Grid Layout**

The CSS grid lets you define a two-dimensional layout of columns and rows and then place elements within the grid. Some elements may only fill one cell of the grid; others can span multiple columns or rows. The size of the grid can be defined precisely, or you can allow it to automatically size itself as needed to fit the contents within. You can place items precisely within the grid, or allow them to flow

naturally to fill in the gaps.

grid layout applies to two levels of the DOM hierarchy. An element with display: grid becomes a *grid container*. Its child elements then become *grid items;*

The container behaves like a block display element, filling 100% of the available width. Although not

shown in this listing, you could also use the value inline-grid; in which case, the element

will flow inline and will only be as wide as is necessary to contain its children. You’ll most likely not use inline-grid as often;

* Finally, the grid-gap property defines the amount of space to add to the gutter between each grid cell.
* grid-template-columns and grid-template-rows. These define the size of each of the columns and rows in the grid. This example uses a new unit, fr, which represents each column’s (or row’s) *fraction unit*. This unit behaves like flex-grow factor in flexbox;

## **Anatomy of a grid**

* *Grid line*—These make up the structure of the grid. A grid line can be vertical or horizontal and lie on either side of a row or column.
* *Grid track*—A grid track is the space between two adjacent grid lines; rows or columns.
* *Grid cell*—A single space on the grid,
* *Grid area*—A rectangular area on the grid made up by one or more grid cells.

Note: I find it’s generally easier to build a page from the outside in;

### **Numbering grid lines**

With the grid tracks defined, the next portion of the code places each grid item into a specific location on the grid.

|  |
| --- |
|  |

You can use the grid numbers to indicate where to place each grid item using the grid-column and grid-row properties. If you want a grid item to span from grid line 1 to grid line 3, you’ll apply grid-column: 1 / 3 to the element. grid-row: 3 / 5 to a grid item to make it span from the horizontal grid line 3 to grid line 5.

NOTE These properties are in fact shorthand properties: grid-column is short for grid-column-start and grid-column-end; grid-row is short for grid-row-start and grid-row-end.

You can also specify grid-row and grid-column using a special keyword, span This tells the browser that the item will span one grid track.

### **Working together with flexbox**

They each shine in different scenarios. Choosing between flexbox and grid for a piece of a design is going to come down to your particular needs.

* Flexbox is basically one-dimensional, whereas grid is two-dimensional.
* Flexbox works from the content out, whereas grid works from the layout in.

flexbox is one-dimensional, it’s ideal for rows (or columns) of similar elements. Grid, on the contrary, is two-dimensional. It’s intended to be used in situations where you want to align items in one track with those in another;

With grid, however, you are first and foremost describing a layout, then placing items into that structure. While the content of each grid item has the ability to influence the size of its grid track, this will affect the size of the entire track and, therefore, the size of other grid items in the track;

We’ve positioned the main regions of the page using grid because we want the contents to adhere to the grid as it is defined. But for some other items on the page, such as the navigational menu, we can allow the contents to have a greater influence on the outcome; that is, items with more text can be wider, and items with less text can be narrower. It’s also a horizontal (one-dimensional) layout. For these reasons, flexbox is a more appropriate solution for these items.

When your design calls for an alignment of items in two dimensions, use grid. When you’re only concerned with a one-directional flow, use flexbox. In practice, this will often (but not always) mean grid makes the most sense for a high-level layout of the page, and flexbox makes more sense for certain elements within each grid area. As you continue to work with both, you’ll begin to get a feel for which is appropriate in various instances.

## **Alternate syntaxes**

There are two other alternate syntaxes for laying out grid items: named grid lines and named grid areas. Choosing between them is a matter of preference.

### **Naming grid lines**

Sometimes it can be a bit tricky to keep track of all the numbered grid lines, especially when working with a lot of grid tracks. To make this easier, you can name the grid lines and use the names instead of numbers;

grid-template-columns: [start] 2fr [center] 1fr [end];

This declaration defines a two-column grid with three vertical grid lines named start, center, and end.

You can then reference these names instead of the numbers when placing grid items in your grid.

grid-column: start / center;

This declaration places a grid item so it spans from grid line 1 (start) to grid line 2 (center).

You can also provide multiple names for the same grid;

grid-template-columns: [left-start] 2fr [left-end right-start] 1fr [right-end];

In this declaration, grid line 2 is named both left-end and right-start. You can then use either of these names when placing a grid item.

You can use named grid lines in countless ways. How you use them can vary from one grid to the next, depending on the particular structure of each grid.