Table of Contents

[Cascade 3](#_Toc142730310)

[Understanding stylesheet origin 3](#_Toc142730311)

[USER AGENT STYLES 3](#_Toc142730312)

[IMPORTANT DECLARATIONS 3](#_Toc142730313)

[Understanding specificity 4](#_Toc142730314)

[INLINE STYLES 4](#_Toc142730315)

[SELECTOR SPECIFICITY 4](#_Toc142730316)

[A NOTATION FOR SPECIFICITY 4](#_Toc142730317)

[SPECIFICITY CONSIDERATIONS 4](#_Toc142730318)

[Understanding source order 4](#_Toc142730319)

[LINK STYLES AND SOURCE ORDER 5](#_Toc142730320)

[CASCADED VALUES 5](#_Toc142730321)

[Two rules of thumb 5](#_Toc142730322)

[Inheritance 5](#_Toc142730323)

[Special values 5](#_Toc142730324)

[Using the inherit keyword 5](#_Toc142730325)

[Using the initial keyword 5](#_Toc142730326)

[Shorthand properties 6](#_Toc142730327)

[Beware shorthands silently overriding other styles 6](#_Toc142730328)

[Understanding the order of shorthand values 6](#_Toc142730329)

[Summary 6](#_Toc142730330)

[**Working with relative units** 6](#_Toc142730331)

[**The power of relative values** 6](#_Toc142730332)

[The struggle for pixel-perfect design 6](#_Toc142730333)

[**Ems and rems** 7](#_Toc142730334)

[**EMS FOR FONT SIZE TOGETHER WITH EMS FOR OTHER PROPERTIES** 7](#_Toc142730335)

[**THE SHRINKING FONT PROBLEM** 7](#_Toc142730336)

[Using rems for font-size 7](#_Toc142730337)

# 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;